Copy No. _____

Defence Research and Development Canada
Recherche et développement pour la défense Canada

DEFENCE R&D DÉFENSE

An Introduction to Software Agents

Tania E. Randall

Defence R&D Canada – Atlantic

Technical Memorandum

DRDC Atlantic TM 2007-221

February 2008

Canada

An Introduction to Software Agents

Tania E. Randall

Defence R&D Canada – Atlantic

Technical Memorandum
DRDC Atlantic TM 2007-221
February 2008

Principal Author

Original signed by Tania E. Randall

Tania E. Randall

Approved by

Original signed by Francine Desharnais for

Jim S. Kennedy

Head, Maritime Information and Combat Systems

Approved for release by

Original signed by C. V. Hyatt

C. V. Hyatt

DRP Chair

Abstract

This paper provides the reader with a general background on software agents, covering the main topics found in the agent literature. It consolidates pertinent information from a wide variety of sources into a single comprehensive document. Its aim is to provide enough information to educate those unfamiliar with agents, and guide them in deciding whether or not agents are an appropriate modelling tool for their own work.

Résumé

Le présent document donne au lecteur un contexte général sur les agents logiciels, qui couvre les principaux sujets trouvés dans les documents sur les agents. Il regroupe, en un seul document complet, les données pertinentes provenant de toute une gamme de sources. Son but est de fournir suffisamment d'information pour former ceux qui ne connaissent pas les agents et de les diriger pour qu'ils puissent décider si les agents constituent un outil de modélisation approprié à leur travail.

This page intentionally left blank.

Executive summary

An Introduction to Software Agents

Randall, T.E.; DRDC Atlantic TM 2007-221; Defence R&D Canada – Atlantic; February 2008.

Background: Over the past few years, the Virtual Combat Systems (VCS) group at Defence R&D Canada – Atlantic (DRDC Atlantic) has been developing a virtual maritime environment for command and control (C2) level, human-in-the-loop experimentation using the Virtual Maritime Systems Architecture (VMSA). While human operators have been used to control the friendly (blue) force entities, the opposing (red) force entities have been controlled by the simulation. Most recently, the red force entities have been modelled within the Joint Semi-Automated Forces (JSAF) environment. JSAF entities do have some reactive intelligence (e.g., they avoid collisions with other entities, and fire weapons at blue forces when within range) and are able to follow instructions from the JSAF operator (e.g., follow paths, travel to a specific location, and pause), however, they do not make decisions for themselves or take proactive steps to get themselves closer to their ultimate goals (such as ‘destroy a high valued unit’). Having red forces which exhibit realistic behaviours is critical to the success of an experiment as well as the credibility attributed to it by military operators participating in the experimentation program. Agent-based technology has previously been used to model autonomous, reactive, and proactive entities and may be applicable to modelling red force entities for VMSA. This paper provides an overview of software agents and represents the first step in the investigation of agents for this purpose.

Results: This paper provides an overview of agents, how they can be used, and how they are implemented. An agent is an autonomous software entity that can react to and act on its environment. It may also exhibit other characteristics, such as being proactive, which results in the entity not only reacting to the events that occur within its environment, but also acting based on its own internal desires. Among other things, an agent may also exhibit an ability to learn from past experience and communicate and cooperate with other agents in order to achieve shared goals. Agents are a good modelling choice when the behaviours that must be modelled are not easily predicted. For example, agents are well-suited for modelling human decision making or systems controlled by human decisions; it also seems reasonable to use agents to enhance the behavioural fidelity of platform entities in an existing simulation such as VMSA.

Significance: This paper provides motivation for using agents to provide behavioural control of maritime entities in the VCS Group’s virtual environment. Agents have great potential to significantly increase the realism of any non-human controlled entities in future simulations, thereby increasing the simulation’s credibility and its potential for generating accurate experimental results.

Future plans: The JACK agent development toolkit has been purchased and will be used to construct agents to control simulated red forces, starting with basic behaviors and working up to more complex behaviours. A contract is in place to have JACK integrated with the existing virtual environment such that agents will provide the decision making for the computer controlled entities and the existing simulation will model their sensors, weapons and physical characteristics.

Sommaire

An Introduction to Software Agents

Randall, T.E.; DRDC Atlantic TM 2007-221; R & D pour la défense Canada – Atlantique; février 2008.

Introduction : Ces dernières années, le Groupe des systèmes de combat virtuel (SCV) de R & D pour la défense Canada – Atlantique (RDDC Atlantique) a développé un environnement maritime virtuel pour l'expérimentation au niveau commandement et contrôle (C2) avec la participation de personnes réelles, en faisant appel à l'architecture de systèmes virtuels maritimes (VMSA). On a eu recours à des opérateurs humains afin de commander les entités de la force amie (bleue), alors que les forces ennemies (rouges) étaient commandées par la simulation. Tout récemment, les entités des forces rouges ont été modélisées au sein de l'environnement des forces interalliées semi-automatisées (JSAF). Les entités JSAF ont une certaine intelligence réactive (par exemple, elles évitent des collisions avec d'autres entités et font usage de leurs armes contre les forces bleues lorsqu'elles se trouvent à portée de tir) et sont en mesure de suivre des instructions de l'opérateur JSAF (par exemple, suivre des trajets, se rendre à un endroit précis et faire un arrêt), mais elles ne prennent pas de décision d'elles-mêmes ou ne prennent pas de mesure proactive pour se rapprocher de leurs objectifs ultimes (comme « détruire une unité de valeur élevée »). Le fait d'avoir des forces rouges ayant des comportements réalistes est essentiel au succès d'une expérience, ainsi qu'à la crédibilité que lui attribuent les opérateurs militaires qui participent au programme d'expérimentation. La technologie fondée sur des agents logiciels a déjà servi à la modélisation d'entités autonomes, réactives et proactives et pourrait s'appliquer à la modélisation d'entités rouges aux fins de l'architecture VMSA. Le présent document donne un aperçu des agents logiciels et représente la première étape de l'examen des agents à cette fin.

Résultats : Le présent document donne un aperçu des agents, de la façon de s'en servir et de la façon de les mettre en œuvre. Un agent est une entité logicielle autonome qui peut réagir à son milieu et influer sur son milieu. Il peut aussi faire preuve d'autres caractéristiques, comme se montrer proactif, ce qui a pour effet de lui permettre de non seulement réagir aux événements qui se produisent dans son milieu, mais aussi d'agir en fonction de ses propres désirs internes. Il peut notamment faire aussi preuve de la capacité d'apprendre de l'expérience et de communiquer et de coopérer avec d'autres agents pour atteindre des objectifs qu'ils partagent. Les agents représentent un bon choix pour la modélisation lorsque les comportements qu'il faut modéliser ne sont pas facilement prévisibles. Par exemple, les agents conviennent bien à la modélisation de la prise de décisions humaine ou de systèmes contrôlés par des décisions humaines; il paraît aussi raisonnable de s'en servir pour améliorer la fidélité de comportement d'entités de plate-forme dans une simulation en place, comme l'architecture VMSA.

Portée : Le présent document donne des raisons de se servir d'agents dans le but de commander le comportement d'entités maritimes dans l'environnement virtuel du groupe SCV. Les agents offrent un excellent potentiel pour améliorer considérablement le réalisme d'entités commandées non humaines dans des simulations futures, ce qui accroît la crédibilité de la simulation et les possibilités qu'elle offre pour générer des résultats expérimentaux précis.

Recherches futures : La trousse d'outils de développement d'agents JACK a été achetée et servira à la construction d'agents pour commander des forces rouges simulées, en commençant par des comportements de base et en passant à des comportements de plus en plus complexes. Un marché est en place en vue de l'intégration de la trousse JACK à l'environnement virtuel existant, pour que les agents assurent la prise de décisions pour les entités commandées par ordinateur, et que la simulation en cours modélise leurs capteurs, leurs armes et leurs caractéristiques matérielles.

This page intentionally left blank.

Table of contents

List of figures

List of tables

This page intentionally left blank.

1 Background

Over the past few years, the Virtual Combat Systems (VCS) group at Defence R&D Canada – Atlantic (DRDC Atlantic) has been developing a virtual maritime environment for command and control (C2) level, human-in-the-loop experimentation using the High Level Architecture (HLA)-based Virtual Maritime Systems Architecture (VMSA) [1]. In one way or another, these experiments have dealt with the operator's ability to detect and identify hostile entities and their follow-on actions. Identification of an entity is primarily based on its behaviour. In our simulations, however, the modelling of entity behaviour has been somewhat crude. In the beginning, these computer controlled entities (known as computer generated forces (CGFs)) were nothing more than a specific type of entity placed on a game board prior to run-time with a series of waypoints to follow at specified velocities. Regardless of what might be happening around them (for example, a human-controlled entity firing at it or heading directly for it), the entity performed as initially planned. In later experiments, the semi-automated forces software known as JSAF (Joint Semi-Automated Forces) was used to control both neutral and red force entities. With this, entity behaviour became more sophisticated, with the ability to have an entity follow a path, travel to a point, pause, etc., and automatically avoid other entities if necessary, resuming their previous actions afterwards. When given the proper permissions, these entities would also fire upon opposing force entities when in range. The red (and neutral) force game controller could also change the actions of these entities on the fly by modifying destination points, path plans, speeds, etc. Setting up a JSAF scenario, however, took considerable time and patience as JSAF is neither user friendly or intuitive. The increase in effort required to set up a JSAF scenario did not result in a proportional increase in simulation realism. Beyond this, there were limits to the types of information that could be shared between JSAF and VMSA, and JSAF entities could not be controlled by external sensors or simulations. These shortcomings spawned a desire for alternative methods to set up and control the computer generated forces (CGFs) for VMSA simulations.

CGFs should be able to 'think' (or appear to think) for themselves, reacting appropriately to situations on the fly and even making proactive decisions to meet their goals (such as 'destroy the enemy' or 'defend self'). The more realistic the behaviour of these entities, the more credible the overall simulation and the more meaningful the results of experiments that involve these entities.

Agents were identified as a prime candidate for providing such behaviour. While the research field of behaviour modelling extends well beyond the use of agents, and the use of agents extends well beyond their use for behaviour modelling, you'd be hard pressed to find a course or conference on behaviour modelling that didn't include multiple applications involving agents. The purpose of this paper is to provide the reader with an overview of the field of agents. While the intent is to keep the paper quite general in nature, the intended application (i.e., the use of agents to model the behaviour of entities in a maritime simulation environment) has influenced its focus and inspired the examples provided where possible.

2 What is an agent?

First things first. What exactly is an agent? It would be nice to provide a clear-cut answer to this question, but sadly such an answer does not exist, and if attempted, would no doubt be criticized by many others who have tried to answer the question in such a concise way. Below is a short selection of the many agent definitions that can be found in the literature:

> "An agent is anything that can be viewed as perceiving its environment through sensors and acting upon that environment through effectors." [2]
>
> "A software agent is a computational system which has goals, sensors, and effectors, and decides autonomously which actions to take, and when." [3]
>
> "An autonomous agent is a system situated within and a part of an environment that senses that environment and acts on it, over time, in pursuit of its own agenda and so as to effect what it senses in the future." [4]
>
> "An agent is a computer system that is situated in some environment, and that is capable of autonomous action in this environment in order to meet its design objectives. […] an agent is an autonomous entity that is usually goal driven, meaning that all of its actions will be done in accordance with a specific goal, pursued by the agent. The actions applied by the agent are therefore a response to a percept or a sequence of percept that defines the agent's view of its environment." [5]

It is generally agreed that an agent has an awareness of its environment and that it has the ability to react to it and act on it. It is also generally agreed that the definition of an agent includes its ability to act on its own (autonomously) without user input, though the definition from [4] tends to suggest that not all agents are autonomous. The final three of these definitions mention or allude to the agent having goals that it is trying to meet, and suggest the agent may make proactive (rather than simply reactive) decisions towards actions in order to meet these goals.

In the end, it may not really matter which definition is used, or what meaning is attributed to the word 'agent', only that the chosen meaning is clearly defined when it is used.

There are various attributes that may be applied to agents. It is the existence or absence of each of these characteristics that help nail down the type of agent that is being talked about. For the purposes of this paper, a minimalist approach will be taken and a basic agent will be defined to be nothing more than:

> "… a software entity that can act autonomously on and in response to its environment".

That is, an agent necessarily has the characteristics of being autonomous and reactive. An agent can certainly have additional characteristics (in fact, all of the interesting ones do), but these will be identified by using the appropriate adjectives or terminology that will be later defined. For example, "a flexible, cooperative and rational agent" is autonomous, reactive, flexible, cooperative and rational.

It is written in [6] that “software agents, like people, can possess different levels of competence at performing a particular task”. Software agents, particularly when it comes to simulation environments, are often used to represent people or the decisions they make, so with this in mind, it makes sense that agents exhibit a wide range of characteristics, just as people do.

2.1 Attributes of agents

This section provides a (non-exhaustive) list of commonly referred to agent attributes, and gives a short definition of each.

It is assumed in this paper that an agent is both autonomous and reactive:

1. **Autonomous -** the ability to operate (select tasks, prioritize, make decisions, etc.) without human input; and,

2. **Reactive -** the ability to perceive changes in its environment and react in a timely fashion.

In addition to being autonomous and reactive, agents may have any combination (or none) of the remaining attributes.

The following attributes refer to the agent’s intelligence or sophistication:

3. **Pro-active / goal-oriented -** the ability to take initiative and make decisions to act based on an inherent desire for the world to be a certain way;

4. **Inferential capability -** the ability to act on abstract task specifications; to ‘know’ beyond the information given; and,

5. **Adaptive / learning –** the ability to change its behaviour based on its previous experience.

and the following refer more to the agent’s personality:

6. **Veracious -** the agent will not knowingly communicate false information;

7. **Benevolent -** the agent does not have conflicting goals, and will therefore always try to do what is asked of it; and,

8. **Rational -** the agents will not act in a way that will prevent its goals from being achieved.

A simulation may consist of single agent, but more likely it will consist of multiple agents. The following attributes are only applicable in a multi-agent system:

9. **Social ability -** the ability to communicate with humans and/or other agents via some kind of agent communication language (possibly resembling human-like ‘speech acts’);

10. **Cooperative –** the ability to work with other agents in order to achieve a common goal and/or perform actions that help another agent achieve its goals; and,

11. **Collaborative -** the ability to work with other agents through both communication (social ability) and cooperation.

The final two characteristics that will be mentioned here are more related to the capability of the software representing the agent than the agent itself and should be transparent to anyone observing a simulation of such agents:

12. **Persistence –** the ability of an agent to maintain its identity and state for long periods of time, and possibly over run-time failures; and,

13. **Mobility -** the ability of an agent to transport itself from one machine to another for the purposes of sharing the load over multiple machines, reducing network traffic, accessing data, etc..

From combinations of these characteristics, other 'catch' phrases are developed. For example, the terms ***intelligent*** agent and ***flexible*** agent suggest that an agent is situated in an environment and able to react to changes in it (i.e., satisfies the basic definition of agent-hood), and is also pro-active and social.

A ***weak*** agent is also identified as one with (at a minimum) the properties of autonomy, social ability, reactivity, and pro-activity. In addition to satisfying these characteristics, a ***strong*** agent is considered to have ***information attitudes*** (information about the world in the form of beliefs and/or knowledge) and ***pro-attitudes*** (which guide the agent's actions, such are desire, intention, or commitment). In other words, strong agents possess qualities that are more often thought of in terms of humans than software [7]. ***Strong*** agents may also be referred to as ***intentional*** agents.

The beliefs-desire-intentions (BDI) agent model is a commonly referred to example of an intelligent agent with both information attitudes (beliefs) and pro-attitudes (desires and intentions). Intentional agents, based on the BDI agent model, will be used for modelling the behaviours of CGFs in the VMSA simulation environment. BDI agents are discussed further in Section 4.

Beyond the issue of whether or not an agent exhibits a particular characteristic is the extent to which it exhibits it and how it is implemented by the agent designer. This is addressed to some degree in Section 7.1 which discusses variations in implementations of internal architectures.

2.2 Classification of agents

There is no one classification scheme for agents. Depending on the purpose of the classification, it may be broken out in many ways:

- By the (combination and/or degree of) attributes that that the agents exhibit (e.g., weak / strong agents, problem solving ability, autonomy, intelligence (reasoning or learning), mobility, or planning);

- By their purpose or role (e.g., e-mail filtering, surveillance, personal assistant, human decision maker);
- By their implementation:
    - using a symbolic reasoning model (referred to as *deliberative*), or a stimulus/response type of model (referred to as *reactive*) ;
    - by their control architecture (e.g., neural network, algorithmic, rule-based, planner, etc.);
    - by the location of control: central control versus distributed control; or,
    - by the language in which they are written; and,
- By their environment (e.g., database, network, simulation).

From this point on, the phrase 'agent' should be thought of to imply 'intelligent agent' unless otherwise stated.

2.3 Agents versus objects

It is not uncommon to be thinking at this point that object-oriented programming should be sufficient for modelling agent programs, but there are indeed some significant differences betweens agents and objects. From [8], three main differences are as follows:

- Agents are autonomous. In the case of objects, the decision about whether or not an action will occur is determined by the requesting object (the one invoking a method). In the case of agents, it is up to the agent that receives the request for action to decide whether or not it will perform it;
- Agents are smart. Agents exhibit flexible (reactive, pro-active and social) behaviours. They can have multiple goals as well as various ways of achieving these goals; and,
- Agents are always active. An object oriented program with many objects may have only a single thread of control. In comparison, each agent has its own thread of control which is always active.

2.4 When to use agents

All modelling is an abstraction of reality. Determining the correct level of abstraction depends on the problem that needs to be solved. In [7] Dennett suggests that there are three primary levels of abstraction that can be used to predict the actions of a system. He refers to these as 'predictive stances'.

The three predictive stances, in order of increasing level of abstraction, are referred to as the:

- physical stance;
- design stance; and,

- intentional stance.

The physical predictive stance is used when the actions of the system are best predicted using physical concepts. For example, to predict where a baseball will land, it would make sense to consider things like mass, velocity, and the force with which it was hit. This prediction is based on the laws of physics and is generally pretty accurate.

When the system is too complicated to explain using the physical stance, either the design stance or intentional stance must be considered. The design stance is used when the actions of an entity are most easily described by looking at what the entity was designed to do, i.e., what its functions are. For example, a wall thermostat can be used to turn the heat up or down. It is not necessary or useful to model the thermostat circuitry or thermometer coil. All that matters is the thermostat's function; by turning the knob to the right, the heat in the room is increased, and by turning it to the left, the heat in the room is decreased.

The intentional stance is used when the actions of an entity can not easily be explained by physics or the functions of the system. This is generally true when modelling humans or other living organisms and their thought processes. It would be impossible to come up with a complete list of all of the functions that a human could perform, and even harder to model the human thought process with physical formulas. Human actions are better explained by attributing beliefs, knowledge, intent, desires, and goals, etc., to the person and using these to predict which decisions should be made. The intentional stance can be applied to non-living entities in some cases, as well as to other living creatures. For example, if birds *believe* there is food in a feeder, their *goal* to not be hungry tells them to go and eat from the feeder (i.e., to take *action* to satisfy their goal).

While the intentional stance *could* be used to model systems such as the thermostat mentioned above (e.g., the thermostat has a 'desire' to make the room warmer, forms an 'intent' to heat up the room until it 'believes' the room is warm enough), nothing would be gained by doing this and there is nothing natural about describing it this way.

Each of these abstractions can lead to reasonable predictive models of behaviour when handled correctly. In choosing which stance is most appropriate for the problem at hand, the one which provides the most reliable prediction of behaviour with the simplest implementation is the best solution.

In [7], the following statement is made:

> "An agent is a system that is most conveniently described by the intentional stance; one whose simplest consistent description requires the intentional stance".

There have been others who disagree with this statement; however, its authors are two of the leading agent researchers in the world.

Thus, while agents can be used in many situations, it is precisely when the intentional stance fits the problem that agents should be considered as a modelling solution.

For the modelling of intelligent red or neutral forces in a simulation, it boils down to being able to predict what a particular entity (e.g., ship) will do in a particular circumstance. In general, the entity is controlled by humans, so to predict the entity's behaviour it is necessary to predict how the humans in control make decisions and select actions. Since the thought processes of humans must be modelled, the intentional stance is the most appropriate choice, suggesting that using software agents for modelling CGF behaviours is indeed a reasonable thing to do.

2.5 Why use agents?

Agents make a good modelling choice because of their natural fit with thinking beings. The BDI agent model "calls upon the mentalistic notions of beliefs, desires and intentions from folk psychology, as abstractions to encapsulate the hidden complexity of the inner functioning of an individual agent. Folk psychology is the name given to everyday talk about the mind and the vocabulary we use to explain ourselves such as: beliefs, desires, intentions, fears, wishes, hopes. As people, the use of such language gives us an efficient method for understanding and predicting behaviour [9]". By modelling "the way that we think", it makes it easier to design the model in the first place and perhaps more importantly, to explain the model to invested clients. If the client is able to understand the model and have confidence in it, more faith will be placed in the results.

On the more technical side, agents have the inherent capability to model and interact with their environment, and they have a decision making process in place to guide their choices which may be influenced by other agents in addition to the environment. These things (decision-making engine, communication protocols for agent interaction, etc.) are built into most agent-building software environments, thus saving the developer the time and effort of programming such capabilities from scratch. The developer's primary job is to program how the agent will react in a particular situation.

2.6 Properties of the environment

Recall the definition that was proposed in Section 2 as the minimum requirement for an agent: "a software entity that can act autonomously on and in response to its environment". The environment is a critical component of an agent system, and the properties of that environment greatly influence the ease (or lack thereof) with which the agent system can be developed. The agent literature typically identifies five properties of the environment. Environments may be accessible or inaccessible, deterministic or stochastic, episodic or sequential, static or dynamic, and discrete or continuous [2].

An environment is considered:

- accessible if the agent's sensors are capable of determining everything about the environment that is relevant to the agent (i.e., agent does not need to maintain an internal state to keep track of the world);
- deterministic if, based on the current state and action taken on that state, it is known for certain what the new state of the environment will be;
- episodic if each action the agent takes does not depend on past actions nor influence future actions (i.e., the agent does not need to remember its past or think ahead);

- static if the environment does not change while the agent is making a decision (i.e., agent does not need to re-check the world's state as it is deliberating); and,

- discrete if the number of possible perceptions and actions are countable and can be completely defined.

An agent that exists in an accessible, deterministic, episodic, static and discrete environment is the easiest to model. Of course, most simulations that attempt to model real-world situations will involve an inaccessible, stochastic, sequential, dynamic and continuous environment.

2.7 Topics addressed in the rest of this paper

So far this paper has attempted to provide the reader with a general understanding of what a software agent is, the characteristics it may possess, its relationship with its environment, and when and why to use them.

The following topics will be addressed in the rest of the paper:

- **Internal Architectures** – the agent's mechanism for making decisions;

- **Beliefs-Desires-Intentions Architecture** – a type of architecture with a decision processing loop that is driven by the mentalistic attribute of intentions;

- **Multi-Agent Systems** – systems with multiple agents which interact with one another in order to solve problems or accomplish goals;

- **Communication Architecture** – the communication model used to allow agents of a multi-agent system to communicate with each other;

- **Agent Development System** – a software program that allows a user to build an agent (and multi-agent) system. These systems may be based on specific internal and communication architectures. Some systems may also allow the user to build in their own models, by providing a language that can be used to program these concepts from first principles; and,

- **Methodology -** a set of predefined techniques (guidelines, heuristics, etc.) to assist the user throughout the life cycle of an agent-based application (including its management) ([10], [11]), identifying expected deliverables at the end of each stage.

This will be followed by a description of some examples of typical agent applications, and a short summary discussion.

3 Internal Architecture

The agent's internal architecture describes how an agent makes decisions. They consider how the agents discussed in the previous section can actually be developed. In general, internal agent architectures can be divided into three high level abstract categories: deliberative, reactive and hybrid.

3.1 Deliberative

A deliberative agent or agent architecture is one whose reasoning is based on symbolic representations and manipulation. The agent contains an internal representation of the environment (its 'world model') and its own mental state. The agent perceives its environment in order to maintain the information in its world model. Decisions, such as action selection, are made through logical reasoning, based on pattern matching and symbolic manipulation [7]. The system can be provided with a general outline for solving a problem and can apply this to the current situation in order to find a solution that was not explicitly stated. The term 'deliberative' is used since these architectures generate a set of all possible actions (in relation to selecting a goal or achieving one) and then choose one of them to perform.

Such architectures are based on the *physical symbol hypothesis* which says that a system made up of:

- symbols and patterns of symbols (representing the physical aspects of the problem);
- symbolically encoded instructions for manipulating those symbols (to find possible problem solutions); and,
- a search capability (to select one of the possible solutions)

is capable of intelligent action [12].

In order to select an action in a deliberative architecture, the following must exist: a complete symbolic description of the world in relation to the problem at hand, a list of actions which can transform one state to another, the set of all states that can be reached given the current state, and a way to evaluate all of the paths leading from the current state to the desired state in order to select the most appropriate one [13]. Thus, a deliberative architecture can be very complicated and time consuming to implement (especially in comparison to a reactive system, as discussed next). As well, since there can be a lot of searching involved in picking the best solution, it is hard for these systems, with any amount of sophistication, to maintain real-time.

3.2 Reactive

A reactive agent or agent architecture is one where actions occur immediately when the preconditions for that action are met. The agent senses the environment in order to determine its state, but it does not maintain any internal representation of the environment's state. There is no thinking or deliberating involved. Preconditions are met and actions are fired – it's as simple as that. Actions may occur in parallel as well, when the current conditions satisfy more than one built-in rule, assuming the actions do not interfere with each other. For example, when touching a hot stove, you may pull your hand away and scream at the same time. Interfering actions that are triggered at the same time may either be combined into a single action or prioritized.

The reactive architecture is based on Rodney Brook's '*Reasoning without representation*' approach [14]. Brooks suggests that intelligent behaviour can be generated without explicit representations (as in traditional Artificial Intelligence (AI) and deliberative systems), and that intelligence is an emergent (rather than built-in) property of certain complex systems.

Reactive architectures are fairly simple to implement and require very little computational time so they can easily run at real-time. This type of architecture can be completely valid for certain circumstances, e.g., insect behaviour and other instinctual behaviour. However, it loses its value when new, unexpected situations occur and there are no rules built-in to handle the situation. If it does not have a rule for that particular situation, no action will occur.

3.3 Hybrid

The hybrid architecture is exactly as the name suggests: a hybrid of the reactive and deliberative architectures. The hybrid architecture is an attempt to combine the reactive and deliberative architectures, in order to benefit from the positive aspects of both types. The hybrid architecture includes a symbolic representation of the world and the ability to reason about it through symbolic manipulation to handle complex situations. However, it also includes a reactive component that allows the systems to react immediately to certain predefined situations (routine tasks), without the need to reason about what to do. In most cases, the reactive component of the architecture will be given priority over the deliberative component so that the situations calling for immediate reactions can be responded to as quickly as possible.

It seems that the hybrid architecture would be the most appropriate choice for developing behaviours for red and neutral force entities for the VMSA simulation. There will certainly be times where a straight forward reaction is required, such as for obstacle avoidance. However, there will also be situations which require more complex reasoning, such as deciding which side an attack should come from based on the current location of other entities, which would be obtained from the agent's beliefs or world model.

4 Beliefs-Desires-Intentions Agent Model

The BDI architecture is perhaps the best-known agent model, which is more specific than one of the three general categories of agent architectures, but yet not specific enough to define exactly how it should be implemented; some flexibility is left to the software designer. It falls partially into the deductive architecture category, with a reduced amount of time and resources required for reasoning through the use of the concept of intentionality. BDI architecture implementations can also have reactive, reflex-like actions (e.g., JACK Teams) in which case they are referred to as hybrid architectures.

BDI agents are situated, reactive, proactive and social. They possess 'mentalistic' attitudes (i.e., beliefs, desires and intentions) and are based on Bratman's theory of practical reasoning:

> "Practical reasoning is a matter of weighing conflicting considerations for and against competing options, where the relevant considerations are provided by what the agent desires/values/cares about and what the agent believes." [15]

The meaning of the agent's mentalist attitudes vary somewhat from author to author, but for the purposes of this document:

- **beliefs** represent the agent's knowledge of the world (including of itself and other agents);

- **desires** are the world states that the agent has interest in achieving. Individual desires may not be realistic and they may conflict with other desires (e.g., 'destroy entity x' and 'remain undetected');

- **goals** are a subset of the desires that are both realistic and consistent (e.g., 'destroy entity x' *or* 'remain undetected'.);

- **intentions** represent the goals that the agent has committed to achieving by choosing to execute a plan; and,

- **plans** are a series of actions that lead the agent and/or its environment from one state to another (i.e., satisfy an intention).

It is the intentions that drive an agent to action. Intentions persist (must be fulfilled or deemed impossible to fulfill), constrain future actions (e.g., actions which hinder satisfying a current intention will not be taken), and must seem achievable to the agent. Intentions thus reduce the time that an agent must deliberate over what to achieve next, since it will immediately drop any options that would hinder the satisfaction of current intentions.

Practical reasoning involves two main processes: figuring out what needs to be achieved (what the intentions are) and then figuring out how to achieve them (which plan(s) should be executed). Both of these processes take time and computational resources. They can not be performed indefinitely, so bounds must be set and at some point the processes must be ended, even if the solutions are not optimal.

Plans include the following elements:

- a goal which the plan intends to satisfy;
- a pre-condition list (what must be true to use the plan);
- a delete list (things that are no longer true after executing the plan);
- an add list (things that become true by executing the plan); and,
- a body (a list of actions to perform).

From [15], the basic BDI processing loop looks like this:

```
while true
    observe the world
    update the internal world model
    deliberate about what intention to achieve next
    use means-end reasoning to find a plan to achieve these intentions
    execute the plan
end while
```

Figure 1: The generic BDI processing loop

The process is illustrated in [16], and a slightly modified version of it is presented in Figure 2.

When it comes time to implement a BDI system however, the process may be more complicated than the loop suggests. The system may also address how committed the agent will be to its particular intentions. In some, an agent may reconsider its intentions when a new event occurs, whereas in others it may only drop an intention if it believes it has been achieved. The way that commitment issues are handled is not dictated by BDI theory and is a matter of choice on the part of the developer.

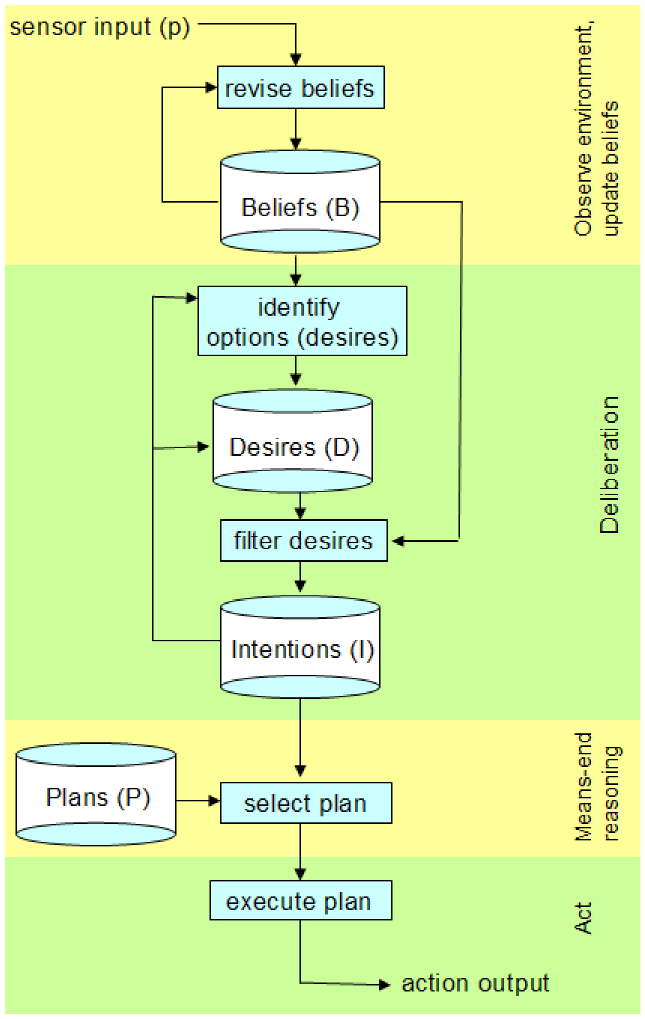

Figure 2: Illustration of the generic BDI processing loop

5 Multi-Agent Systems

A system is considered ‘multi-agent’ if it is made up of autonomous entities that demonstrate the following characteristics [17]:

- each agent has incomplete capabilities to solve a problem;
- there is no global system control;
- data is decentralized; and,
- computation is asynchronous.

In a multi-agent system, agents interact with one-another. The agents may share the same goals or have conflicting or disjoint goals. The agents may cooperate with each other to achieve shared goals, coordinate their plans, negotiate with each other, compete with each other in the case of conflicting goals, etc. Essentially, these agents do all the things that people would do in order to meet their goals.

Figure 3 is a commonly referenced diagram of a multi-agent system that originated in [18].

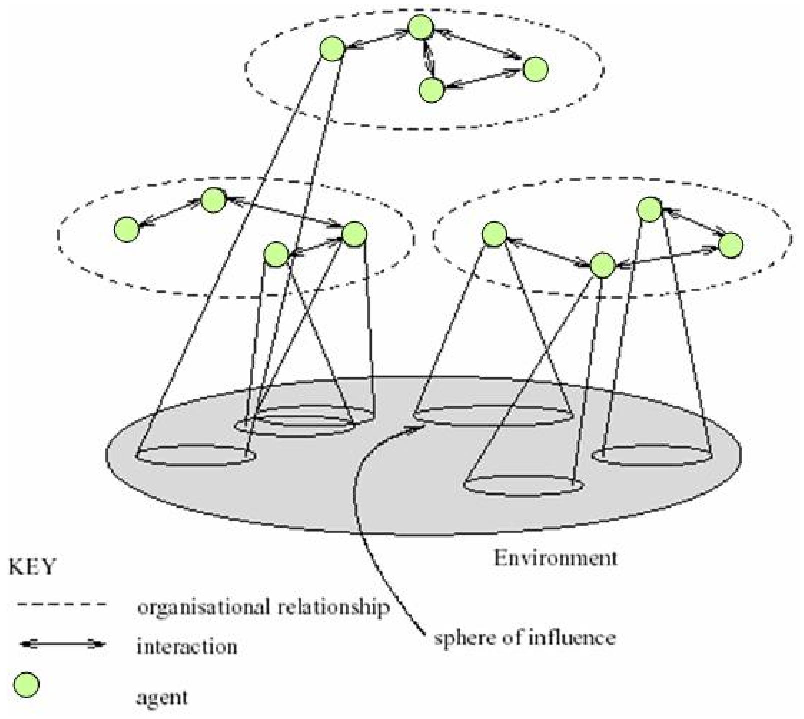

Figure 3: Multi-agent system

This diagram illustrates that a multiple agent system, where some agents interact with each other, forming ‘organizational’ relationships. Also, each agent has the ability to influence and understand only a portion of the environment, which may coincide with the ability of another agent.

The real-world is a multi-agent environment. People can not achieve all of their goals without consideration for others in the world.

5.1 Why use MASs?

There are many reasons why one might choose to model a system using a multi-agent system. However, the main reason is to be able to solve problems that are too complicated to be solved by a single agent with limited capability, knowledge and computational resources. When multiple agents work together and bring together their own knowledge, resources, etc., solving larger problems becomes more feasible. As well, there is less risk than there is with a single centralized agent; there’s no single point of failure. Another reason one might choose to model with a multi-agent system is that the system being modelled is naturally viewed as a system of interacting entities (for example, fishing boats in a fishing village). Multiple agents can also be used to make efficient use of spatially distributed resources.

5.2 Single versus Multi-Agent Systems

It should be clarified that it is possible to have a model with multiple agents that is not really a ‘multi-agent’ system. An environment is only considered multi-agent if the entities in the system must work together or compete with each other in order to achieve their goals. If this is not the case, from the point of view of an individual agent, all other agents are just part of the environment.

The agents that will control the behaviours of red force entities for the VMSA simulation environment could be thought of as a multi-agent system if the red forces will be working together as a team to accomplish their goals (e.g., destroying a high-valued unit). They could also work together to build a picture of their environment through sharing tracks with each other. However, they could also be modelled using a single agent system with multiple agents by providing each agent with access to all of the information it will need through its own sensors. For example, it could be sufficient for the agent to base its choices on the things it can independently see, e.g., which other agents it can see and how damaged the target appears.

5.3 Working with other agents

In order for agents to be able to work together, they must be able to coordinate with each other. Coordination may occur in a variety of ways. Three methods for coordination as identified in [15] follow.

5.3.1 Joint Intentions

The theory of ‘joint intentions’ is referred to as a model of teamwork, and describes a way in which agents can coordinate. It highlights the importance of intentions to the coordination process (much as they are the cornerstone of importance to the reasoning of a BDI agent). From [15], intentions “provide both the stability and predictability that is necessary for social interaction, and the flexibility and reactivity that is necessary to cope with a changing environment.”

Being a part of a team implies that the agent feels some sense of responsibility towards its shared goals. In order for a team to commit to trying to achieve something (e.g., to destroy entity x), they have to believe two things: (1) that that ‘something’ has not already been achieved (e.g., entity x is not currently destroyed) and (2) that that ‘something’ is achievable (e.g., that they can destroy entity x). Then, once this joint commitment has been made by the team, it can only be dropped if all team members *mutually* believe it has been achieved, is unachievable, or is no longer valid (e.g., a higher priority target has appeared). If one agent decides that the goal is met, unattainable or no longer valid, it will inform all of the others of this rather than simply dropping the intention and no longer participating in the team effort.

5.3.2 Partial Global Planning

In this method, agents share information with each other in order to achieve a common understanding of the problem they are trying to solve. This problem is one that can not be solved by any individual agent and cannot be solve with any single plan (which is where the term ‘partial’ comes from). The term ‘global’ is used because the agents coordinate with each other in order to come up with an understanding of the problem that is beyond the ‘local’ view of any single agent. The process of partial global planning involves three iterated steps as described in [15]:

- Each agent develops short-term plans to achieve its own goals;
- Agents share the details of their plans with each other and determine where their plans overlap; and,
- Agents modify their own plans to better coordinate with the other agents while still achieving their own goals.

5.3.3 Coordinated Distributed Problem Solving

This teamwork-based model of coordinated distributed problem solving involves four steps [15]. First, an agent must recognize that it either cannot solve a problem on its own (e.g., due to lack of resources or capabilities) or does not want to solve the problem on its own because it believes that solving it collectively would be better in some way. For example, a small boat armed with a rocket propelled grenade launcher would unlikely be able to destroy an enemy frigate on its own, but with ten similarly armed team mates, the chances for success are greatly increased. As well, the agent may figure that by acting with the team, its own chances of survival are increased. In this recognition stage, the agent must also believe that there are agents out there that can help achieve the goal. For example, the agent must believe there are other small armed boats available to participate in the attack.

During the next step, the agent that wants help in achieving a goal solicits help from the other agents. If the solicitation is successful, this step ends in a group of agents jointly committed to achieving a goal (e.g., destroying the frigate).

In the third step, the agents come up with a plan for achieving the goal. There may be more than one plan that will move the agents closer to their overall goal (e.g., begin launching grenades as soon as boat is in range, or begin launching grenades once the entire team is in range), so they have to agree on which plan will be used. Reaching such agreements can involve the use of techniques like negotiation, voting or auctions as described in the next section. The final stage involves the execution of the plan chosen in the previous step.

5.4 Competing with other agents

When reading about multi-agent systems, there will typically be references to an agent's ability to participate in negotiation, auctions, and voting. These techniques are used in multi-agent systems to help agents reach agreements. While it is often not explicitly stated in the literature, it seems that the agreements that the agents must reach may be in regard to which goal should be achieved first, which plan should be used to get the agents collectively closer their overall goal(s), or which agent should be responsible for which task. The processes become particularly important when the agents are self-interested rather than benevolent, meaning that they are not satisfied by decisions that benefit the overall system while hampering their own individual goals (i.e., they don't believe in 'the greater good'). While it is not clear that these processes will be required for the implementation of the red forces, at least in its initial phases, a brief overview of negotiation, auction and voting techniques used by agents is provided below.

Negotiation is used by agents to reach agreement on matters of common interest. Agents make proposals, review other agent's proposals and subsequently modify their own proposal until an agreement is reached. This process usually involves a series of rounds, where each agent exposes its proposal in turn. Each agent has its own private strategy for making proposals. Each negotiation process is monitored by a rule that determines when a deal has been made amongst the agents and what the details of the deal (agreement) are.

The agents in the red force modelling application could use negotiation techniques to decide when and where they will gather to discuss their plans. The chosen time and place will have to take into account where the boats currently are and how long it will take them to get to the proposed gathering point. Negotiation strategies might be used to come up with a fair solution for all agents.

Auctions may also be used for the allocation tasks amongst bidders. There is an auctioneer agent and multiple bidder agents. Generally, the auctioneer's goal will be to minimize the cost of the contract (on behalf of the contracting agent who must pay the winner) and the bidders' goals will be to maximize the price that will be paid for the task. Auctions can also be used for the allocation of goods amongst bidders, in which case the auctioneer is more likely trying to maximize the selling price and the bidders are trying to minimize it. There are numerous types of auctions and protocols, including sealed bid, English, Dutch and Vickrey auctions. See Annex A for more information on these auction types.

To stretch the red force modelling example somewhat, the agents could be bidding on what role they will play in the attack (for example, a spotter, a decoy, or an attacker) or which weapons they will be allocated.

Voting techniques allow decisions to be made such that every agent's opinion counts. There are various forms of voting techniques that can be used to select a particular outcome, including non-ranking voting, ranking voting, approval voting, and Coomb's method. See Annex A for more information on these voting schemes.

Red force agents could vote on which attack plan should be executed.

In order to cooperate or compete with each other, agents must have a way to communicate (Section 6).

6 Communication in MAS

In order for agents to be able to work together, they must be able to communicate. If the agents are simply sharing data, it may not be necessary to deal with an Agent Communication Language (ACL). However, if agents will be sharing plans, intentions, and the like, an ACL will likely be required. Without communication, it is only possible for agents to influence each other through changes to their shared environment. Changes can be sensed by another agent, and as a result, the second agent may act differently than it would have without this change. Agents can also affect each other's state more directly, for example, a boat crashing into another boat.

This section will focus on the case where agents need to be able to communicate directly. The most common way for agents to communicate is by passing messages to each other. In [19], a list of criteria for an agent communication language is identified. It states that an ACL should keep the communicative acts (e.g., tell) separate from the content of the message that is being communicated (e.g., the ID of track5 is HMCS Halifax). It turns out that the popular ACLs do indeed include two components: an outer language which describes what to do with the content of the message, and an inner language which describes the message content itself. This distinction is inspired by linguistic 'speech act theory' which tries to explain how people use language to communicate. 'Speech acts', as they are referred to in speech act theory, have two components: a performative verb (e.g., inform) and propositional content ("the contact is a fishing trawler").

A well-known example of an outer language is the Knowledge Query and Manipulation Language (KQML) ([20],[21]) and of an inner language is the Knowledge Interchange Format (KIF) [22]. Using standard languages are preferred, since this allows agents created by other developers to, in theory, be interoperable. KIF and KQML are discussed briefly here.

KIF provides a syntax for the message content which is fairly human-readable without the use of an interpreter. KIF also provides methods to represent meta-knowledge, reasoning rules, objects, functions and their relations.

Here's an example of the KIF language.

> "The identification of track1 is HMCS Halifax" can be expressed in KIF by:
>
> (= (identification track1) (HMCS Halifax)).

KQML allows KIF messages to be passed from one entity to another, assuming they both have the ability to write and read KQML. KQML defines a variety of performatives that can be used to indicate what the receiving entity is supposed to do with the information that it is receiving. Performatives can be used to ask questions, share information, subscribe to information, etc. Here are a few specific examples:

- "ask-if" allows the sender to ask the receiver if the content of the message is in its knowledge base;
- 'tell' allows the sender to tell everyone that the content of the message is in its knowledge base;

- ‘insert’ allows the sender to tell the receiver to add the content of the message to its own knowledge base;

- ‘subscribe’ allows the sender to tell the receiver that it wants to know about any changes to the content of the message; and,

- ‘error’ indicates that the sender thinks the message it received previously from the receiver was (or contained) a mistake.

For a complete list of KQML performatives, see Figures 3 and 4 in [23].

KQML goes beyond this list of performatives, however. It must also, at a minimum, identify the message’s sender and receiver. This information (and possibly more) is included in the KQML message, through the use of KQML keywords. Table 1 shows all the KQML keywords and their meanings. This table is based on the table found in [23].

Table 1: KQML keywords

Keyword	Meaning
:sender	the sender of the message
:receiver	the intended recipient of the message
:from	who the original message was from if it is a forwarded message
:to	the final destination of the message if it is a forwarded message
:in-reply-to	identifies the previous message that this message is in response to
:reply-with	the label that should be used in the ‘in-reply-to’ field of any messages that are sent in response to this message
:language	the language that is used in the ‘content’ field
:ontology	the ontology that is used in the ‘content’ field
:content	the information / content of the message

Figure 4 provides an example of a KQML message, complete with KIF-formatted content.

```
(insert
        :sender agent1
        :receiver agent2
        :language KIF
        :ontology navy
        :content (= (identification track1) (HMCS Halifax))
)
```

Figure 4: Example of KQML/KIF message

The term ‘ontology’ seems to find its way into many areas of research these days, and it certainly serves an important purpose when it comes to agent communication. An ontology, for the purpose of agent communication, is a formal description of all the terms and expressions that can be used by the communicating agents, with detailed descriptions and expected use of each. In order for the agents to be able to understand each other, they not only have to be speaking the same language (e.g., KIF), but they have to be using the same vocabulary (which is identified by the ontology).

7 Examples of Agent Architectures

7.1 Variations in Agent Architectures

In Section 3, three very broad categories of internal architectures were identified and discussed. However, when it comes to actually implementing these architectures, there are many possible variations, resulting in a large number of agent architectures that can be used for building agents. In many cases, developers decide that it is just too complicated to figure out exactly what has been implemented in a particular agent architecture, or to modify known elements of an architecture so that it will work the way the project requires. Developers may decide it would be easier to start from scratch and build exactly what is needed, thereby resulting in even more agent architectures to choose from. [24] is a rather elaborate website that discusses the many elements of an agent architecture and how they vary across architectures. The following list of architectural variations is cropped and summarized from [24] to give the reader an idea of the types of issues being referred to:

- how is knowledge represented, maintained and stored, or does it even store knowledge at all?

- how much knowledge can be stored, how efficiently is data accessed, and how is inconsistent knowledge dealt with?

- what can be learned by the agent (if anything at all) and how? Does the agent learn everything (even when it could decrease performance) or is it selective in its learning?

- how coherent and consistent is the agent's performance? Would the agent make the same decision again given the same situation?

- what methods of planning are supported (e.g., means end analysis[1], forward planning,[2] or backward planning[3])? Can the new plans be saved (learned) so that there is less processing next time?

- can the agent make changes to its plan based on changes in the environment?

- does the architecture have reasoning built in to select which goal to achieve when there is more than one goal? What if there's more than one plan to achieve a goal – is there built-in reasoning for this?

[1] where differences between the current and goal states are used to propose operators which reduce the differences
[2] where, given an impasse, the architecture tries to walk forward through the problem space to the goal
[3] where, given an impasse, the architecture looks at which operators will ultimately achieve the goals and walks backwards to the current state

- what type of reasoning does the architecture support (e.g., does the architecture support deductive[4] reasoning?)?

- if the architecture has a world model, how consistent is it with the real world? Can it be used to predict future states of the world? Can the agent be queried about its state or knowledge of the world?

- is the agent's behaviour coherent[5], salient[6], and adequate[7]?

- how does the architecture deal with an impasse (i.e., a situation where a decision cannot be made because there isn't enough information available)? Does it have a way to handle this, does the system crash, or does it do nothing at all?

- does the agent assume its sensing capabilities are perfect (i.e., that the environment is *accessible*)?

- can the system guarantee real-time performance? What effect does this time constraint have on the agent's ability to perform rationality?

- is the processing interruptible? Can a high priority event trigger an agent to suspend its current activity to attend to the event?

- is the architecture scalable (can it handle complex problems)? How does it perform over time?

- is the architecture organized modularly (where functional components are separated from one another), hierarchically (where capabilities of lower levels are inherited from the levels above), both (where each level of the hierarchy is divided into functional components) or neither?

To this list, we will add the following that were not mentioned in this website:

- can agents communicate with each other, and if so, how?

- does the architecture have built-in support for specific agent models, such as the BDI model?

- is it possible to integrate this architecture with other simulations, or is it intended to create stand-alone applications only?

- what coding language is used to develop the agents?

[4] anything that can possibly be derived from the facts in the agent's knowledge base is automatically added to the knowledge base.
[5] the agent can resolve conflicts between competing goals
[6] the agent understands its current situation and can use that comprehension to act appropriately
[7] Ability to select actions to take in an appropriate order to meet a goal. For example, if Block A is on top of Block B, in order to put Block B on top of Block A, it would be necessary to first take Block A off of Block B.

- are the architectures designed for real-world applications or for simulation?

Before looking at some specific architectures, it should be possible to make a few statements of a more general nature. Any architecture supporting purely reactive agents will not support planning or reasoning. It will not maintain a world model and will do nothing if an impasse is encountered. Real-time performance can be expected of a reactive agent system, due to the fact that there is limited processing involved (no reasoning or planning, just trigger conditions triggering actions). Both deliberative and hybrid architectures will have some type of knowledge representation and world model, as well as planning and reasoning capabilities. Real-time performance is of greatest concern for purely deliberative architectures, and of less concern for hybrid architectures.

Unfortunately, the website that the list above was summarized from [24] is quite outdated, and while it is reasonable to assume that the content above is still valid, only two of the three architectures (i.e., the Subsumption Architecture and the SOAR Architecture) that we will be looking at more closely in the rest of this section are considered by the website authors; neither JACK, nor any of its predecessors are mentioned. Still, this website seems to be one of the few places where architecture characteristics are discussed and then related to particular architectures. It is much easier to find the inverse, i.e., documentation on an architecture (or comparison of architectures) that discusses its (their) characteristics. However, even in that case, architectural documentation tends to focus more on the decision cycle implemented or the tools the architecture provides for the user, and not get into the more specific details. So, since the Subsumption Architecture and Soar have been addressed by the website, it is fairly easy to provide the reader with an idea of how the items in this list apply to those architectures (though it may not account for changes made to Soar in recent years). However, in the case of JACK, discussion here is limited to the information available in the JACK documentation, which doesn't necessarily detail each of these things. So, in the discussions that follow within each specific architecture section, lack of a comment in regards to a particular element *does not* mean that the architecture does not address or support that feature. It simply means that documentation stating information about that feature could not be found with respect to that architecture.

The Subsumption Architecture is a purely reactive architecture. The Soar architecture primarily uses deliberation to handle the information coming in from its percepts, but it is possible to create seemingly reflexive actions for some percepts, so perhaps Soar could be referred to as a hybrid system, though for the most part it is deliberative. Finally, JACK is of the hybrid variety and provides clear support for the BDI model (though it can support other models as well).

An important note is that while the Subsumption Architecture is considered here in the same context as Soar and JACK, this may be a bit misleading. Soar and JACK are both software systems that can be used to build agent systems. The Subsumption Architecture is really more of an agent paradigm (at the level of a BDI agent system). However, it is not clear that there are any well-know software environments for developing Subsumption Architectures (perhaps they are not needed, or have not gained the same amount of popularity, since the system is relatively simple), though [25], [26] both claim to allow users to build agents built on a Subsumption Architecture. Still, in keeping with the common usage of the term "Subsumption Architecture", we continue to consider in the same context as Soar and JACK.

The following sections will discuss these three architectures in more detail.

7.2 Subsumption Architecture

The subsumption architecture is a common example of a reactive architecture. It is based on Rodney Brook’s theory from the mid 80s that intelligence did not need to be built into the system (such as in deductive systems), but rather could emerge from the system. He argued that insects exhibit forms of intelligent behaviour (finding food, avoiding objects, etc.) which couldn’t even be closely modelled by robots (the real-world agents) of that time. He figured that scientists were over-analyzing the problem, and that there must be a simpler way to obtain these types of behaviours.

In this architecture, complex behaviours are broken down into simple behaviour components and organized into layers, each handling a particular goal of the agent. The behaviours at the lower level are more primitive than those at the top layer. All layers can access sensor data and contribute to action selection. The overall goal of the agent is handled by the top layers, whereas the common sense / reflex goals are typically handled at the lower levels. Since the subsumption architecture was originally invented for the control of robots, robot movement makes for a natural example of the architecture. A goal such as ‘find food’ will be at a high layer than ‘avoid obstacle’, since while pursuing the goal of finding food, the goal of avoiding an obstacle will temporarily override the food finding goal if there is an object in the robot’s path.

Such systems can generate very reasonable behaviour at very low computational cost. In fact, robots built using the subsumption architecture exhibited much higher forms of intelligence than any other robots at that time, even though the system was much simpler. However, like all reactive architectures, systems built based on subsumption concepts do not allow for any explicit reasoning and are unable to deal with unexpected situations.

In terms of the list in Section 7.1, Subsumption has both a modular organization (each module controlling a low level task such as a sensor) and a hierarchical organization (each level produces a behaviour, and one behaviour may be overridden by another of a higher level). It is interruptible in the sense that it is always reacting to the information obtained through its sensors, and at any time new information may trigger a new behaviour. It does not have a world model of its own – it uses the real world as its model. Subsumption Architecture agents are considered coherent because they are constantly receiving information from their sensors and making decisions based on them. They are salient, as long as every relevant sensor is modelled, so that the system is not missing any critical information, in which case it will not be able to react to the changes that could have only been detected by such a missing sensor. Since Subsumption responses are limited to those of a reflexive nature, agents can easily maintain real-time. In terms of Subsumption architectures, scalability may be an issue in that there may be a limit to the number of layers an architecture can have and it may not be possible to produce truly complex behaviours, however at the time of writing of this website, these limitations had not yet been seen. Since Subsumption architecture does not use symbolic logic, symbolic logic can not be learned by the agent. However, Subsumption was originally built for the navigation of robots, and these robots are able to build a map of the area they are moving through. In a sense, this suggests learning. However, the agent’s (robot’s) ability to do this was explicitly hard coded in the agent program itself. Clearly, since all actions of Subsumption agents are of a reflexive nature, Subsumption has no intention of supporting a BDI model, or any other agent model requiring similar reasoning capabilities.

7.3 Soar

Soar is based on the *Unified Theory of Cognition* and is referred to as a cognitive modelling architecture. The idea behind the unified theory of cognition was to take all the evidence that had been gathered and theories that had been accepted over many years of research in all the different fields related to human cognition (psychology, biology, logistics, etc.) and come up with a unified theory that would satisfy all disciplines. Typically, each of these disciplines is looking at a different part of the problem, but in some cases they overlap. It isn't necessary that the theories underlying one part of cognition also be used to explain another, but that as a whole, the theories were at least compatible with one another. Soar is an example of an architecture that has tried to achieve such an ambitious objective.

Soar tries to model all of the functionality required in a human brain, at a very small granularity, in order to generate reasonable actions and, in particular, an ability to learn. For this reason, the internals of Soar are fairly complicated and this can also be true of Soar programs (which can require thousands of rules). However, the efforts are repaid with good models of human intelligence. Soar's goal is to be able to reasonably approximate complete rationality and general intelligence.

While Soar users have made attempts to map Soar to a BDI architecture [27], the mapping is not obvious or straightforward, and is unlikely to achieve any additional understanding of intelligence modelling. However, it is not necessary to use BDI architectures to build powerful models.

Soar is a symbolic artificial intelligence system, and more specifically is based on production systems, presumably because of their ability to perform a wide range of searches. Production systems are made up of a database (short-term memory), production rules (if-then else statements leading to actions under the right conditions) and a control system (which is responsible for deciding which production rules to use and in which order).

Soar has two types of memory: working memory, which represents the current situation (sensor information, current goals, etc.), and long-term memory, which contains instructions for handling the different situations that are detected by the working memory.

Problem solving in Soar is achieved by searching through the currently achievable system states and choosing the operator that is going to move the system closest to its goal state (the solution to the problem) [28]. Choosing each move is quite an involved process. *Figure 5*, recreated from [28], shows a diagram of Soar's decision cycle.

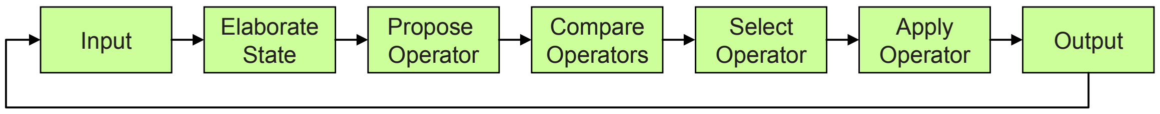

Figure 5: Soar's decision cycle

Elaborating state, proposing an operator, comparing operators, and applying an operator all require Soar to access information in long-term memory and pull it into working memory. Elaborating the state involves taking information in the working memory in combination with long-term memory and adding any information to working memory that can be inferred from the combination of the two so that it is readily available to the selection functions. Operators identified as applicable in the given situation are then proposed. These operators are compared and one is chosen through the use of preference weightings and a decision procedure. This operator is applied, causing a change of state (output). This cycle repeats and eventually, through a series of moves, the final goal state is achieved.

It is possible, however, that moving through the problem space will not be quite so straightforward. Sometimes it is not clear which state will best move the system closer to its goal. In this case, the system is said to be at an impasse. Soar's ability to handle this situation is one of its unique and powerful features. In this case, Soar creates a new goal to solve the impasse. Soar uses a variety of techniques to solve impasses, including means-ends analysis and hill climbing [29]. Once a solution is found, Soar uses a technique referred to as 'chunking' to create a new rule that can be used the next time the same situation is encountered. Thus, it won't be an impasse the next time, and the system has learned.

Soar uses Tcl for programming. However, a variety of software products have been built to assist the user in building Soar programs.

Soar does not provide any functions for building models of teamwork. However, models of teamwork can and have been created in Soar. For example, STEAM [27], [30] was written in Soar (with approximately 300 rules) and is based on Joint Intention theory. STEAM introduces the idea of team operators (rather than just operators as in Soar) which are used to represent actions that can be performed by a team rather than individuals.

In terms of the list in Section 7.1, Soar has a hierarchical organization with levels representing components of the human 'architecture': biological, cognitive, rational and social. It is interruptible in the sense that when it reaches a point where it can no longer proceed (an impasse), it creates a new sub-goal of dealing with the impasse, and shifts its attention to handling that sub-goal, before returning to the main goal. Soar models the worlds symbolically through productions. Every time an agent reaches an impasse, it creates a new production so that that situation won't result in an impasse the next time. In this sense, Soar learns and the size of Soar's knowledge base may grow to be very large. While limits on how big it can get don't seem to be of great concern, the time it takes to search the constantly growing knowledge base can be. Soar is not selective in what it learns – it 'learns' everything, whether or not it might be useful in the future (it does not have any meta-level reasoning to help it determine whether or not to maintain the new information). In terms of planning capability, when Soar encounters a problem, it tries to find a series of actions (a plan) that will take it from its current situation to its goal state. While executing a plan, if new and relevant information becomes available through its percepts which modify its goal, Soar is able to re-plan to meet the new goal. Soar has been mainly used for simulated environments, but has also been applied to real-world applications (like Robots).

7.4 JACK

JACK is a software environment for designing, building, and running MASs. JACK's architecture is of the hybrid variety and provides considerable support for the BDI reasoning model. JACK also allows developers to implement their own reasoning models if desired. JACK promises easy integration with other software using standard protocols, such as High Level Architecture (HLA), and JACK Sim (a JACK extension) promises the ability to run *repeatable* agent-based simulations. JACK Teams (another JACK extension) handles the formation, execution and coordination of teams. Co-JACK (yet another extension) offers the ability to model influencers such as tiredness, fear, and other emotions and investigate their affect on the agent's performance or actions. JACK is very 'lightweight' and can handle thousands of entities on low-end computer systems.

JACK is intended to control the actions selected by entities (agents), but not necessarily to implement these actions. It is best used in coordination with existing software that already has the physical models built and requires additional intelligence for decision making. For example, JACK has been used to provide decision making for entities built in the United States (US) Army's OneSAF simulation, but the actions of the entities (e.g., movement, shooting, sensors) are modelled within OneSAF [31]. It is choosing where to move, what and when to shoot, how to react to new sensor information, etc.: that is JACK's niche. However, since JACK is just an extension of JAVA, the physical models can certainly be built within JAVA and used with JACK, but the JACK language and tools will not make representing these physical models any easier than using JAVA alone.

JACK includes various tools to assist the developer in creating and debugging agent-based models and simulations. It includes a development environment, a component browser and editor, a design tool which generates shell code from diagrams, a plan editor where plan reasoning can be expressed by non-programmers using diagrams, a plan tracer to see what part of a plan is currently active during run-time and to see the values of variables in a plan, and an agent interaction display to trace inter-agent communication [32].

Another significant component of the JACK software is the JACK Agent Language (JAL). JAL extends JAVA through syntactical additions, a compiler to convert JACK code into JAVA code, and the JACK *kernel* for run-time support. More specifically, JACK offers the following extensions to JAVA [33]:

- keywords to represent agent components (e.g., agent, plan, event, beliefsets);
- attributes for agent components (e.g., information for beliefsets);
- statement to represent static relationships (e.g., which plans for what events);
- statements to manage the agent's state (e.g., update beliefs);
- a compiler to convert JAL into pure Java code;
- the kernel which provides the default behaviour of agents in reaction to events, plan failures, etc., as well as the communication infrastructure for MASs;
- automated concurrency management;
- default agent behaviours (e.g., to deal with failure of a plan); and,
- native, light-weight communications infrastructure for multi-agent communications.

JACK claims to provide support for building BDI agents. However, desires and intentions are not directly mentioned in the JACK documentation, nor are they defined when building the agents. JACK agents are defined in terms of beliefsets that they can access, events that they are able to respond to and post, and plans that they can execute [34]:

- JACK **beliefsets** are contained in a generic relational model that it used to manage (store, query, update, delete, etc.) the agent's beliefs about the world. The agent's beliefs may or may not actually be true.
- JACK **event**s may be either posted or handled by an agent. Agents can post events to themselves, as a result of a change in one of its beliefsets, in an attempt to accomplish

one of its goals, or through the execution of one of its plans. Agents also specify which events they are able to handle. If an agent handles an event, it has at least one plan that can be used in response to this event. In JACK, there are two types of events: normal and BDI. Normal events are those which are only meaningful for a moment in time (e.g., the location of a ping pong ball during a game). Agents select the first possible plan to handle these events and if it fails, they do not try again because the event is no longer meaningful. BDI events, on the other hand, have more staying power. In response to BDI events, an agent commits to a certain outcome and continues to try plans until it achieves it (or the circumstances change, making it no longer feasible to do so).

- JACK **plans** describe a series of actions that can be followed in order to achieve a particular outcome (i.e., satisfy a goal and handle an event). As only some plans can achieve a particular goal and a plan is only valid in certain circumstances, the plans contain fields to indicate this type of information which can be used for filtering through the plans to find an appropriate choice for a given event or goal.

JACK also provides **capabilities** which can be used to encapsulate combinations of other components (e.g., beliefsets, events, plans) that the agent can use to achieve something. For example, the capability to 'play ping pong' is much more concise than listing all the events and plans that are required to play ping pong). Capabilities are convenient when it comes to software reuse as well.

JACK's execution cycle follows from the BDI architecture. When a BDI event occurs, the agent identifies which plans are applicable to that event and then from those it identifies which plans are relevant based on the current situation. There may be more than one relevant plan, in which case the agent may simply select the first plan in the list, or the developer can add in other methods to decide between these plans. The chosen plan is then adopted as one of the agent's intentions. The agent executes the plan and while doing so it may be necessary to further deliberate if a new event occurs during its execution that may be of higher priority than the executing intention. If the plan fails, the agent will try another plan from the relevant plan set. This continues until the intention is satisfied, deemed unattainable, or no longer necessary.

JACK Teams allows the agents to form teams and cooperate as teams. A team is considered to have its own beliefs, desires and intentions in addition to the individual reasoning components held by the individual agents. Teams are defined to have particular roles and can achieve things for itself or for other teams (by acting as a sub-team). Behaviours for teams are also specified as plans, or, more specifically, teamplans [35].

In terms of the list in Section 7.1:

- JACK has a modular design and new functionality can be added by writing plug-ins for JACK [32];
- JACK agents are interruptible. Once a JACK agent has chosen a plan, it will work through the steps of a plan until either the current intentions are satisfied or deemed unsatisfiable, or until a new event occurs that prompts JACK into further deliberation [34];
- JACK plans are precompiled procedures that may be selected by an agent based on their applicability and relevance to the current situation. JACK agents do not (seem to) create

new plans, so they are not planners, though they are capable of reasoning in order to choose the most appropriate pre-existing plan;

- JACK agents 're-plan' only in the sense that they may interrupt their current plan and choose a new, more appropriate one, based on new events;
- JACK agents can learn new information (by adding information to their beliefset), but do not learn when to use plans beyond their original capability, unless it is done through the information in their beliefsets. For example, a beliefset could store data on the effectiveness of a weapon against a particular type of target, and the statistics in the beliefset could be used to guide future plan selection;
- JACK's world model takes the form of one or more beliefsets for each agent, containing the information that the agent either knew at start-up or has obtained throughout run-time. Beliefs in a beliefset are stored in the form of first order relational tuples [34];
- Real-time performance is not an issue for JACK – it is very lightweight and it is possible to run thousands of JACK agents on low-end computers [32];
- There may be more than one applicable and relevant plan for handling a particular event under certain conditions, and rather than just selecting the first plan in this set (which is an option), JACK also allows for more detailed reasoning about which plan should be executed. It does this through meta-level reasoning, which essentially means that JACK has additional plans for dealing with selecting amongst various plans;
- JACK has been used for both simulation and real-world purposes. By default, communication in JACK uses a TCP-based protocol, but other protocols can be layered on top of this or replace it entirely (e.g., KQML) [33].

8 Methodologies for Building Agent Systems

Agent methodologies support the development of agent systems by providing a process and guidelines for the developers to follow. There are many multi-agent system methodologies, each with their own approach, making it fairly difficult to know which to choose. However, in general the methodologies stem from three areas: object-oriented (OO) programming, knowledge engineering (KE), and requirements engineering (RE).

The following diagram (Figure 6) from [10] identifies some of the existing methodologies as well as their roots. 'P' denotes a group of methodologies without common roots, except that the methodologies in this category were all inspired by a specific agent platform or architecture.

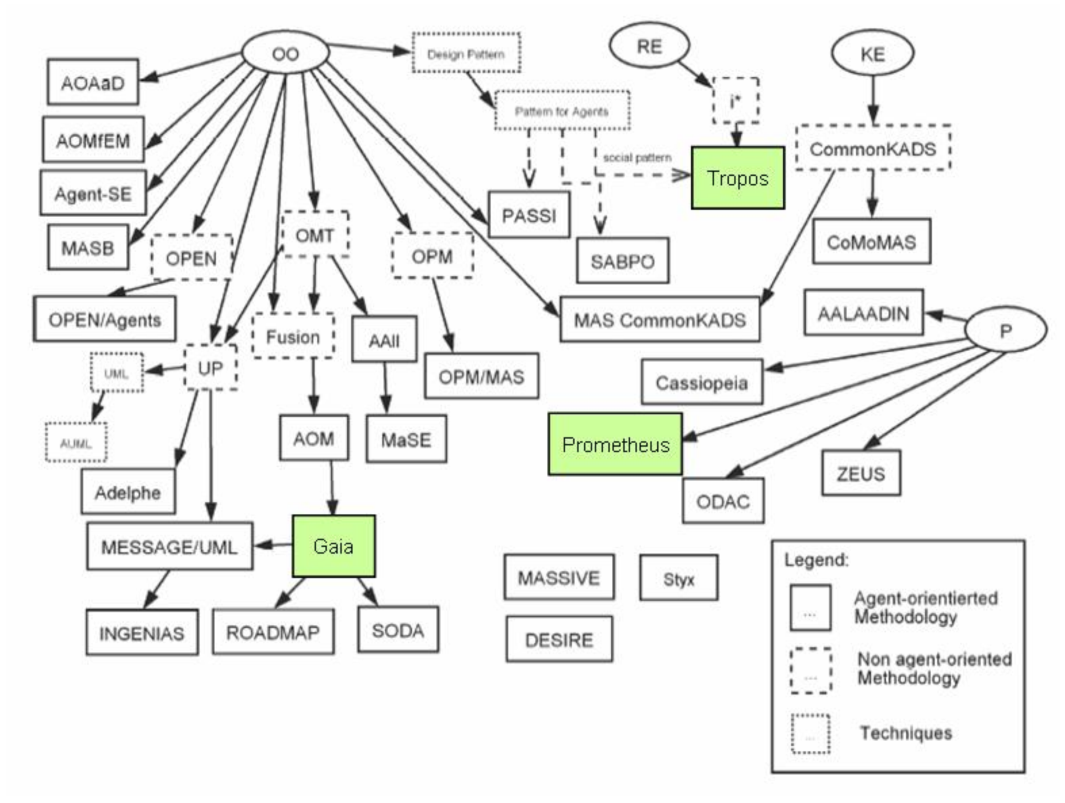

Figure 6: Methodologies for building agent systems

A brief outline of the three methodologies highlighted in green in Figure 6 is provided in the remainder of this section. These methodologies were chosen because they seem to be the ones most commonly referred to (ROADMAP and ZEUS are also popular), and have detailed descriptions of their methodologies available. However, other methodologies may also be applicable even if the exact application has not been identified by the author. As long as the key constructs used in the methodology can be mapped to JACK constructs (agents, beliefs, plans, capabilities, etc.), then the methodology may be applicable.

Each of these methodologies (Prometheus, Tropos and Gaia) guide the user to develop a specification for agents using a top-down approach – they look at the requirements for the system and through a series of steps break these requirements down to identify the specific agents required.

While Prometheus and Tropos cover similar portions of the overall design process (from requirements to implementation), Gaia focuses its efforts on the analysis and design phases of the process. A visual comparison of the three is provided in Figure 7. This figure is a condensed version of a figure in [10].

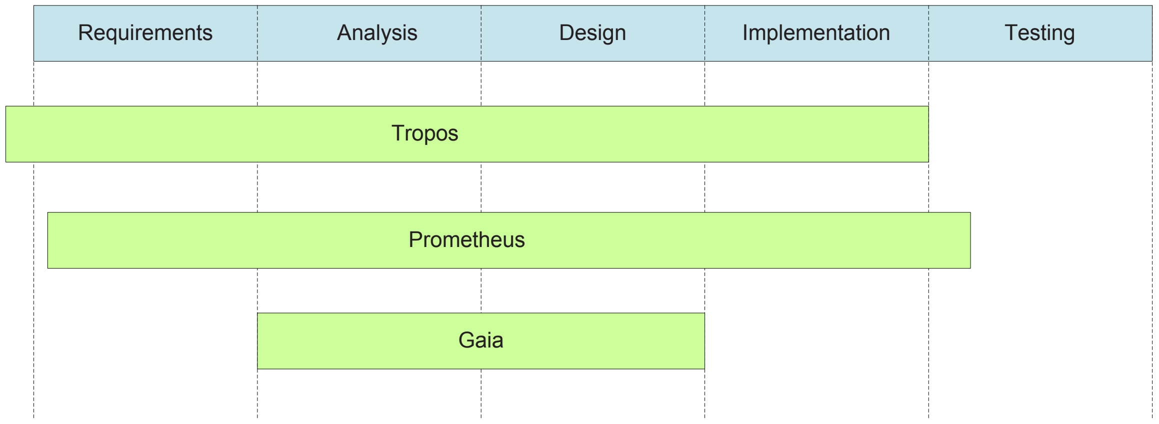

Figure 7: Comparison of agent system development stages covered by three methodologies

8.1 Prometheus

Prometheus was designed for use by people who are not experts in agents. It supports the development of intelligent agents which use goals, beliefs, plans, and events. Some argue a methodology should be more general, but the Prometheus designers argue it is more important that it is useful. Prometheus may be less useful than the others for developing non-BDI agents, but is particularly useful for building BDI agents. The focus of Prometheus is on the internal architecture of the agent(s). The steps to this methodology are as follows [36]:

1. System specification
    a. What information can an agent perceive from its environment and what actions can an agent take that affect the environment?
    b. What are the goals and what functionality is needed to achieve these goals? Identify some use case scenarios (examples of how you foresee the system in use).
2. Architectural design
    a. Define agent types: create a data coupling and agent acquaintance diagram and use them to identify groups of functionalities which become the agent types. For each agent type, identify the following: when is it initialized & destroyed, what are its functions, actions, percepts, goals, data consumed and produced, and events it can handle.

b. Design overall system structure: create a system structure diagram that shows how the overall system will function. It must show the agent types, communication links between them, and data inputs and outputs. Create interaction protocols (using the use case scenarios), defining the valid sequences of messages between agents.

c. Define interactions between agents.

3. Detailed design phase (this phase focuses on the internals of each agent type (capabilities, internal events, plans, and a detailed data structure))

   a. Create capability descriptors which describe/illustrate which events are generated and received, what data is produced and consumed, and any interactions with other capabilities.

   d. Create plan, event and data descriptors.

   e. Create agent overview diagrams.

The JACK Development Environment (JDE) includes a tool with which overview diagrams can be drawn. These diagrams are linked to the model so changes to the diagram are reflected in the JACK code of the model. Prometheus has its own tool as well, the Prometheus Design Tool (PDT) which provides forms to enter design entities, generates overview diagrams and design documents.

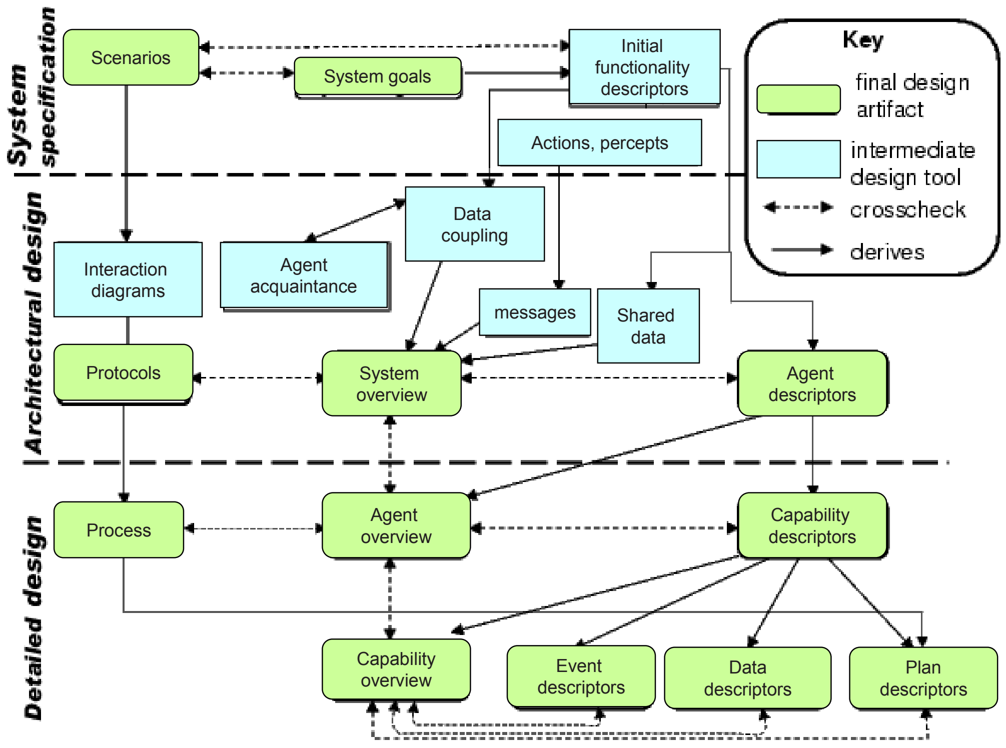

Figure 8: The Prometheus methodology

Figure 8 is a diagram from [36] that provides a visual representation of the Prometheus methodology.

The Prometheus methodology was used in [37] to design a simple path planning agent for an Unmanned Aerial Vehicle (UAV) and in [38] to model a team of gold miners.

8.2 Tropos

Tropos places emphasis on the earlier requirements which have motivated the need to develop the software, and still manages to be a fairly comprehensive methodology that guides the user through to the completion of the implementation stage.

Some terminology definitions may be needed to understand the Tropos stages. First, an **actor** is an entity with strategic goals and intentionality. It is meant to be a term that is more general than 'agent', and includes agents as a sub-class. Second, a **dependency** between two actors implies that one actor relies on another in order to achieve a goal, execute a plan, etc. [39].

From [39], the process is divided into five stages which are defined as follows:

1. Early Requirements: identify stakeholders (actors) and their objectives (goals);
2. Late Requirements: add the system-to-be as another actor and identify its dependencies with the stakeholder actors. Identify how the system interacts with its environment;
3. Architectural Design: add more system actors with goals and tasks which are sub-goals or sub-tasks of the system;
4. Detailed Design: define system actors with more detail, including specifications of communication and coordination protocols; and
5. Implementation: transform output from detailed phase into a skeleton for the implementation. This requires mapping from the Tropos constructs to those of an agent programming platform such as JACK which can be used to implement the agent.

The Tropos methodology was used in [40] to model an integrated health assessment system and in [41] to analyze security requirements of information systems.

8.3 Gaia

The Gaia methodology considers the development of an agent-based system to be a process of organizational design. Gaia breaks the process into two primary steps: analysis and design. It assumes a statement of requirements is available as input to this methodology. The following outlines the Gaia methodology [42]:

1. Analysis phase
    a. Identify goals of overall system and its expected global behaviour.
    b. Describe the environment.

   c. Identify the preliminary roles model (the basic skills required by the organization).
   d. Identify the basic interactions required to accomplish the preliminary roles (without committing to an organizational structure).
   e. Identify the rules that must be enforced by the organization's global behaviour (without committing to an organizational structure).
2. Design phase (uses outputs from analysis phase)
   a. Architectural design phase (identifies the overall architecture of the system with its roles and interactions model).
      i. Identify the system's organizational structure in terms of its topology and control regime, by considering:
         1. the organizational efficiency,
         2. the real-world organization (if any) in which the MAS is situated, and
         3. the need to enforce the organizational rules.
      ii. Complete preliminary role and interaction models (now with commitment to an organizational structure).
   b. Detailed design phase
      i. Define the agent models – map roles (1 or more) to an agent class.
      ii. Define the service model – identify the main activities that the agent will engage in to satisfy its roles.

Gaia does not deal with requirements capturing or with implementation of the specification that it develops. Figure 9 is a diagram from [42] that provides a visual representation of the Gaia methodology.

The Gaia methodology was used in [43] to design a "smart" license management system, in [44] for the development of an airline operations control centre, and in [45] to create a model for bus crew scheduling.

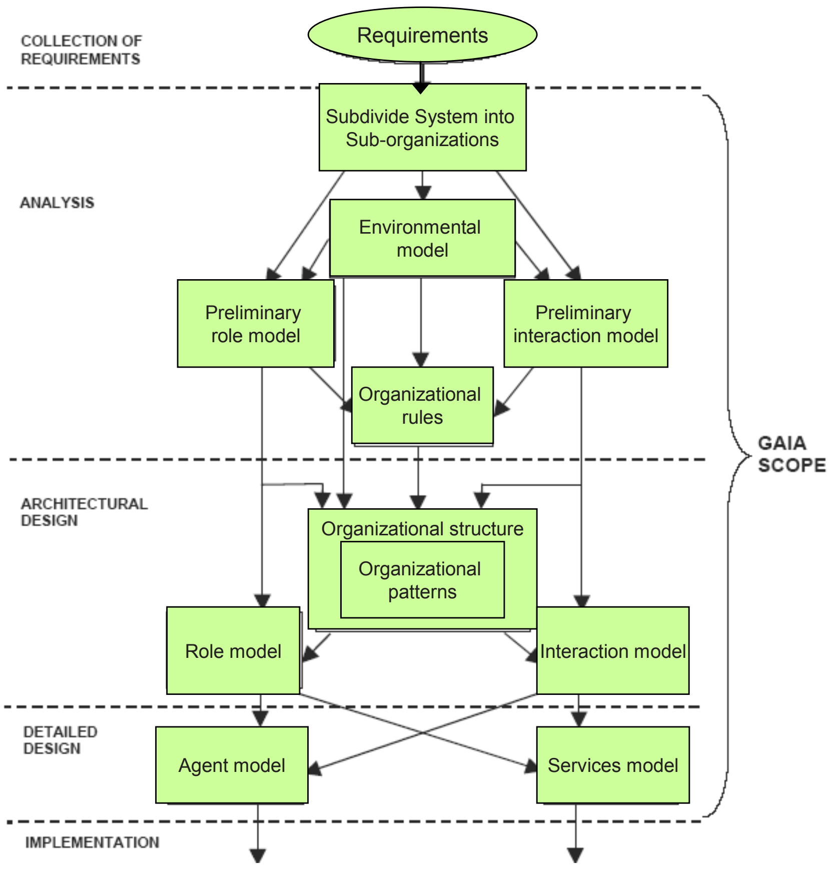

Figure 9: The Gaia methodology

9 Sample Applications

This section provides some examples of how agent technology has been applied in the military domain, with a focus on applications that should be of interest to the VCS group. The discussions will be kept brief and are limited to the information available in the referenced paper. The details of the agent technology used will not necessarily be addressed (primarily because most examples in the literature don't provide such details). It is hoped, however, that they will serve as some concrete examples of how agents can be applied.

The reference for the information provided in each example is given following the title of the example (i.e., the name of the paper).

Example 1: Intelligent Agents for the Synthetic Battlefield [46]

The goal of this research was to remove some of the operator interaction that is required to guide the Modular Semi-Automated Forces (ModSAF) synthetic forces from task to task. The focus was on developing agents for the fixed wing aircraft domain using Soar. A human acted as the battalion commander, but agents were used to represent the company commanders and helicopter pilots. The company commander's actions are deliberative (e.g., he must look at various courses of action, project into the future, etc.). The helicopter pilot's actions are reactive. Helicopter pilots must be able to fly the helicopters and execute their missions autonomously in an uncertain environment. Helicopter pilots work as a team to achieve the goals of the company. The dynamics and states of the helicopters are modelled within ModSAF as are the simulated sensors (vision and radar). This information is made available to the agents via a Soar-ModSAF interface. The agents interact with a Distributed Interactive Simulation (DIS) environment in real-time.

In a typical attack mission, the battalion commander (human) issues an 'operations order' to the company commander (agent). The order states the battalion's mission, what is known about the friendly and enemy situations and specific orders for companies. The company commander analyzes the order and plans a course of action to achieve the mission (taking terrain into consideration). The battalion commander (a human) must approve the company commander's (agent's) plan before it can be sent to the helicopter pilot agents. The helicopter pilot agents read it, identify their tasks and roles in the plan, and begin to execute the mission.

Modelling the teamwork of the pilot agents was an additional goal of this work. Teamwork capabilities were added to Soar (resulting in STEAM) to enable explicit representation of team goals that expand out into goals and plans for individuals.

This agent-ModSAF integration has been successfully applied in various engineering trials. ModSAF operators input battalion operations orders that had been written by subject matter experts (SMEs) and the commander agents and pilot agents were, for the most part, able to handle whatever missions they were assigned.

It is important to note that approximately <u>ten work-years</u> of effort were invested into the development of the pilot agents, commander agent, and the supporting infrastructure.

Example 2: An Architecture to Support Autonomous Command Agents [31]

This paper discusses the development of 'command agents' which are meant to replace a level of the command hierarchy and alleviate some of the work of OneSAF Testbed Baseline (OTB) operators. The OTB operator produces plans that may be used during a battle scenario. The command agents take attack instructions from the operator and produce plans and courses of actions for the entities they control. The command agent receives feedback from the simulation which may influence its future planning. The command agents act according to military doctrine and are aware of the terrain in order for them to plan appropriate routes for the entities. They give an example where a human controls the battalion commander, command agents control company and platoon behaviour (as a whole, e.g., move to a location), and OTB controls individual platoon members (e.g., where each member is, with respect to the others, while they all travel to a location).

The command agents needed to know where the entities under their control (which were created in OTB) were located, as well as their velocity and heading. This information was obtained through a DIS PDU[8]. However, the command agents also needed to know when the entities under their control had detected a target. Target detection was modeled within OTB, but DIS does not carry this type of information. Thus, modifications had to be made to the OTB source code in order to output this information for the external agents. The agents themselves were implemented with JACK Teams.

At the conclusion of the work presented in this paper, JACK-Teams had not yet been fully integrated with OTB. However, work was ongoing and it believed that when completed, these command agents would be able to demonstrate human-like decision making, thereby reducing the requirement for human input to the simulation at run-time.

Example 3: Cognitive Agent-based Approach to Varying Behaviours in Computer Generated Forces Systems to Model Scenarios like Coalitions [47]

The paper looks at enhancing the realism of Computer Generated Forces (CGFs) by assigning them cognitive attributes which vary depending on the situation. These include attributes such as fatigue, fear, stress, temperature, caffeine consumption, etc. that influence the agent's situational awareness (SA), reaction time, memory, etc. These in turn affect the agent's choice of actions (in the form of tactical plans). The impact of the change in one of these psychological attributes is defined by a sensitivity function. The authors point out that current CGF systems try to model what people *should* do as opposed to what they *actually* do. This system is a step towards modeling what really happens.

Cognitive overlays were used to supplement the BDI model with psychological attributes. The implementation used OTB and CoJACK. The system was demonstrated using a scenario involving a team of attack helicopters attacking a ground target, where fatigue influenced the SA of the helicopters which in turn influenced their selection of tactics.

[8] Distibuted Interactive Simualtion (DIS) Protocol Data Unit (PDU)

Example 4: TACOP: A Cognitive Agent for a Naval Training Simulation Environment [48]

In this research, a cognitive agent is developed to act as the opponent (a Tactical Cognitive Opponent (TACOP)) in a specific naval training exercise. Training new operators involves many people beyond those being trained. It requires people to play team members, opposing forces, and an instructor to monitor trainee actions and provide feedback. In the scenario chosen, the student controls the blue task force, including a high value unit that must be protected and needs to get to a certain location. They must decide when and how to split up the task force to accomplish that goal. The tactics for the red force were discussed in detail with a Subject Matter Expert (SME) in regards to the scenario at hand so that they could be programmed into the agents that would represent them. A BDI model was used. Simple beliefs get formed passively through sensor perception (e.g., radar detects something and a belief that a track exists is created). Complex beliefs are formed when the agent reasons (e.g., that a particular track is closest). Desires are formed from the agent's goals - they can be static (e.g., self defence) or dynamic (e.g., shoot at unit). When a desire is in focus, intentions will be created (e.g., desire to attack HVU create an intention to attack). Actions are generated from intentions.

The model was implemented with the COGNET Architecture and Toolset. To create the environment for the agent, COGNET was integrated with VR-Forces of MAK technologies (requiring data-parsers and the development of ontology mappings).

The system was evaluated by two instructors with mixed results. They concluded that they needed more input from SMEs.

Example 5: Controlling Teams of Uninhabited Air Vehicles [49]

This paper describes how a human operator can work with a multi-agent system in order to control a team of UAVs. The operator provides the mission-level guidance and the UAVs self-organize to achieve the goals set by the operator. However, there are certain decisions that are critical enough that ultimately they must be made by the human, so in this case the agents refer back to the human. Other decisions of varying importance may require some input from the human or may be made entirely by the agent. It is predetermined which type of decisions require what level of monitoring. Each type of decision that could have to be made is pre-assigned a value between 0 and 5. A decision with a value of 0 implies that the operator must decide, a 3 implies that the agent suggests something and asks the operator for permission, and a 5 suggests that the agent/system decides on its own.

Their scenario involved four UAVs that were tasked with locating and destroying a high value mobile ground target. The human operator was to act as the pilot of a fighter aircraft that was safely out of range of any attacks that could be made from the ground.

In their set-up, there are multiple types of agents (i.e., not just one agent to represent each UAV). There are 'user' agents which move information from the operator to the system and allocate individual UAVs to specific tasks. There are 'group' agents in charge of the UAVs assigned to a particular task. The group agents know how to plan and coordinate team tasks (based on Joint Intentions theory (Section 5.3.1)). The primary group behaviours are: search for a target, attack a target, fly a route and perform a standoff search. There are also UAV agents (one for each UAV) that send commands to the sensors, weapons, and autopilot, as well as send sensor and state information to the Group agent. Their primary behaviours are: fly a route, take a picture of a ground vehicle, loiter & observe ground vehicle with imaging sensor, search an area with long-range sensors, and release a weapon. Finally, there is a search agent to develop search routes and an attack agent that controls UAVs involved in the attack phase.

While (at the time of this document's publication) it still required multiple operators to control a single real-world UAV, this paper demonstrates a way in which a single operator can control multiple UAVs. This system was evaluated in three Human-in-the-Loop (HIL) trials. It was concluded that the agent-human decision-making partnership was successful overall, though there was a bit of confusion on the part of the agents in the situation where they recommended a decision that the operator decides to reject.

Example 6: Map Aware Non-Uniform Automata [50]

New Zealand's Defence Technology Agency's (DTA) Map Aware Non-Uniform Automata (MANA) falls somewhere in between the agent development system category and the agent application category. MANA [50] can be used to build military-based scenarios composed of military entities (agents) with varying personalities and behaviours, which can make their own decisions. MANA agents are map aware and their behaviours are affected by the events that occur within their environment. MANA's focus is on being able to set up scenarios easily and explore a range of potential outcomes in a small amount of time. It does not include detailed physical models, rather just 'enough' to make the model work. MANA would be well-used to rapidly explore many possibilities in order to focus the attention of a more detailed model or possibly a human-in the-loop experiment, which cannot be so easily re-design and re-tried. MANA agents are each composed of simple behaviours, but when evolving together within one environment, interesting 'global' behaviours may emerge.

MANA has been used to explore the ability of the future United States Army to perform under degraded communications [51]. This study concluded that either a 25% degradation in communication range or significant delays in network communications have significantly negative affects on the army participants. MANA was also used by the Technical Cooperation Program (TTCP) for a study [52] which examined the potential improvements in response to a group of small attack craft with the use of a UAV.

"The MANA model has been used to explore situations as diverse as the impact of training levels, C2, peacekeeping, maritime patrol, and the events in Mogadishu in 1993 described in the book Black Hawk Down" [53].

10 Concluding Remarks

This paper has provided an overview of agents, their architectures, how they can work together, some implementation software, methodologies for designing agent systems, and some examples.

The study was motivated by an interest in seeing if agents would be suitable for modeling the red forces for VMSA simulations. As this does appear to be the case, the possibility will be pursued further. It seems there are a lot of options and therefore choices that will need to be made along the way, though a few choices can be identified up front:

- the agents should be able to handle both reactive (e.g., collision avoidance) and deliberative (e.g., find the enemy) responses/reasoning;

- the beliefs-desires-intentions model will be used to guide the agent's action selections as it is reasonable to assign such characteristics to the sea platforms (which are controlled by humans), the BDI model executes faster than purely deliberative models, and it seems to be the model that information, research, and software tools are most readily available for;

- the JACK software will be used to implement the agents for a variety of reasons, including the fact that it supports BDI modeling, provides tools for assisting with development and testing, is extended through and compiled into pure JAVA code (which is the language the VCS Group uses), can be interfaced with other simulations, etc.; and

- the Prometheus methodology will be followed since it deals very closely with JACK constructs and doesn't require mapping from the final methodology elements to JACK elements.

10.1 Update on VMSA red forces using agents

While this paper remained in draft form, the initial development of agent-controlled red forces for VMSA simulations was completed [54]. The JACK agent software was used to develop agents that can decide how and when the VMSA entities should act. All physical modelling (sensors, weapons, motion, etc.) of these entities remains in VMSA; JACK agents are used as their 'brains' only. The agents can be thought of as replacing the humans that would be controlling these boats in the real world. While first attempts at designing the agent system involved stepping through the Prometheus methodology, it was soon determined that the methodology *for this particular application* seemed to be taking a fairly simple problem and breaking it into unnecessary pieces, only to merge them back together in order to eventually obtain the information that JACK really required, namely, the agents, events, plans, and beliefsets. As it was fairly obvious what each of these elements would consist of, it was decided that the methodology would not be closely followed. It is expected that these agent methodologies could be more applicable to more complex problems, or when the agent development system has not been predetermined, and there is a desire to keep the design more generalized.

It can also be said that no matter how much one might read about agents or the systems used to implement them, a true understanding of how they work can only be obtained through hands-on experience. This paper was written primarily based on a literature review, rather than real-world experience with agents, and while it tries to represent the information collected concisely and accurately, it is recommended that anyone wishing to get involved in agent system design or development should spend at least a little bit of time actually ‘getting their hands dirty’ as this where a true appreciation for agent capabilities, and limitations, will come from. It is hoped that this paper may provide some guidance in terms of understanding the purposes of agents, defining agent and system capabilities that are relevant to a particular project, and ultimately selecting software for implementing agents, as well as a methodology to guide the design process.

Annex A – Examples of Techniques for Negotiation, Auctions, and Voting

Negotiation

A simple and commonly used form of negotiation for the allocation of tasks is the Contract Net Protocol. In this case, agents have individual goals and are self-interested with limited resources for reasoning. When an agent cannot complete a task on its own, it can request help from other agents. It announces the need for help and any interested agents begin bidding on the task which they can either perform themselves or further subcontract to other agents. The 'contract' is awarded to one of the bidders who then takes responsibility for completing that task.

Auctions

Sealed bid auctions: each bidder places a bid without any of the other bidders knowing the details of the bid. Once all bids are in, the contract is given to the cheapest bidder,

English auctions: bids are accepted one at a time and the current bid value is known to all potential bidders. Any new bid must be cheaper than the current value. When no more bids are coming in, the contract is awarded to the cheapest bidder,

Dutch auctions: The contractor announces the price that he wishes to pay for the contract. If there is a bidder (i.e., an agent that agrees to do it for that price) the contract is awarded. Otherwise, the contractor increases the paying price and waits to see if a bid will come in at that price. This continues until a bid is received, and

Vickrey auctions: bids are sealed, and the cheapest bidder wins, but receives the value bid by the second lowest-bidder.

Voting

Non-ranking voting: each agent votes for only one outcome and the outcome with the highest number of votes is chosen,

Ranking voting: the agents rank each possible outcome in order of preference, giving the highest score to the most favoured choice. In 'Borda voting' these preferences are used as weightings which are summed across all outcomes for all agents. The outcome with the highest over ranking is selected,

Approval voting: each agent can vote for as many outcomes as he likes (i.e., once for each outcome that he finds acceptable), and

Coomb's method: each voter ranks all outcomes in order of preference, from lowest to highest. If one of the outcomes is ranked first by the majority of voters, it is chosen. Otherwise, the candidate that is ranked last by the majority is removed and the ranking begins again.

References

Please note: All web links included with these references were active as of June 2008.

[1] Canney, S.A., “Virtual Maritime System Architecture Description Document”, Issue 2.00, Virtual Maritime System Document Number 00034, Defence Science and Technology Organisation, Edinburgh, Australia, July 2002.

[2] Russell, S. and Norvig, P., *Artificial Intelligence: A Modern Approach*, Chapter 2, Prentice-Hall, Inc., 1995.

[3] Millier, M., “Software Agents”, CHI 97 Electronic Publications: Tutorials, Atlanta, Georgia, 1997, available at: http://acm.org/sigchi/chi97/proceedings/tutorial/mm.htm.

[4] Franklin, S. and Graesser, A., “Is it an Agent, or just a Program?: A Taxonomy for Autonomous Agents”, Proceedings of the Third International Workshop on Agent Theories, Architectures, and Languages, Springer-Verlag, 1996.

[5] Halle, S., “Automated Highway Systems: Platoons of Vehicles Viewed as a Multi-Agent System”, University of Laval, Quebec, June 2006.

[6] AgentBuilder website: http://www.agentbuilder.com/Documentation/whyAgents.html.

[7] Wooldridge, M. and Jennings, N., “Intelligent Agents: Theory and Practice”, The Knowledge Engineering Review, Vol. 10:2, pp. 115-152, 1995.

[8] Flores-Mendex, R.A., “Towards a Standardization of Multi-Agent System Frameworks”, Crossroads magazine, available at: http://www.acm.org/crossroads/xrds5-4/multiagent.html.

[9] Goschnick, S. and Sterling, L., “Psychology-based Agent Architecture for Whole-of-user Interface to the Web”, The University of Melbourne, 2002, available at: http://www.solidsoftware.com.au/Information/Paper/Hf2002/Poster-SteveGoschnick.pdf.

[10] Sudeikat, J., Braubach, L., Pokahr, A., and Lamersdorf, W., “Evaluation of Agent–Oriented Software Methodologies – Examination of the Gap Between Modeling and Platform”, Agent Oriented Software Engineering, 2004, available at: http://www.mip.sdu.dk/~bbk/AOSE/AOSE/AOSE04-03.pdf.

[11] Iglesias, C.A., “A Survey of Agent-Oriented Methodologies”, available at: http://www.agent.ai/doc/upload/200302/igle99_1.pdf.

[12] Mangina, E., “Review of Software Products for Multi-Agents Systems”, Applied Intelligence (UK) Ltd., published by AgentLink (www.agentlink.org) , 2002.

[13] Logan, B., “Lecture 7: Designing Intelligent Agents”, School of Computer Science & IT, University of Nottingham, UK, 2004, available at: http://www.cs.nott.ac.uk/~bsl/G53DIA/ .

[14] Brooks, R., “Intelligence Without Reason”, International Joint Conference on Artificial Intelligence, 1991, available at: http://dli.iiit.ac.in/ijcai/IJCAI-91-VOL1/PDF/089.pdf.

[15] Woolridge, M., *An Introduction to MultiAgent Systems*, John Wiley & Sons, Ltd., England, 2002.

[16] Holvoet, T., “Topic 4: Agent Architectures”, Distributed Systems and Computer Networks Group, Department of Computer Science, KuLeuven, available at: http://www.cs.kuleuven.ac.be/~tom/MAS/SLIDES/topic4.agent.architectures.ppt .

[17] Personal notes taken at European Agent Systems Summer School and Autonomous Agents and Multi-Agent Systems, Utrecht, NL, 2005.

[18] Jennings, N., “On Agent-Based Software Engineering”, Artificial Intelligence 117, pp 277-296, 2000.

[19] “Introduction of Ontology and Agent Communication”, e-mail: yunl@idi.ntnu.no, Department of Computer and Information Science,Norwegian University of Science and Technology, obtained from: http://www.idi.ntnu.no/emner/dif8914/essays/Lin-essay2002.pdf.

[20] Finin, T., McKay, D., and Fritzson, R. (editors), “An Overview of KQML: A Knowledge Query and Manipulation Language”, The KQML Advisory Group, March 1992, available at: http://www-ksl.stanford.edu/knowledge-sharing/papers/index.html.

[21] Finin, T., et al, “Specification of the KQML Agent-Communication Language”, Enterprise Integration Technologies, Palo Alto, CA, Technical Report EIT TR 92-04, updated July 1993, available at: available at: http://www-ksl.stanford.edu/knowledge-sharing/papers/index.html.

[22] Genesereth, M.R., and Fikes, R.E. (editors), “Knowledge Interchange Format, Version 3.0 Reference Manual”, Computer Science Department, Stanford University, Technical Report Logic-92-1, June 1992.

[23] Covington, M.A., “Speech Acts, Electronic Commerce, and KQML”, Artificial Intelligence Center, The University of Georgia, Athens, Georgia, June 1997.

[24] Wray., R., Chong, R., Phillips, J., Rogers, S., and Walsh, B., “A Survey of Cognitive and Agent Architectures”, available at: http://ai.eecs.umich.edu/cogarch0/index.html.

[25] Toal, D., Flanagan, C., Jones, C., and Strunz, B., “Subsumption Architecture for the Control of Robots”, University of Limerick, Ireland, available at: http://www.ul.ie/~toald/Publications/IMC-13c.pdf.

[26] Bryson, J., Smaill, A., and Wiggins, G., “The Reactive Accompanist: Applying Subsumption Architecture To Software Design”, The Department of Artificial Intelligence, The University of Edinburgh, UK, available at: http://www.dai.ed.ac.uk/pub/daidb/papers/rp606.pdf.

[27] Schurr, N., “Evolution of a Teamwork Model”, available at: http://www.cs.cmu.edu/~pscerri/papers/SunBookChapter.pdf .

[28] “Soar BICA Research Project”, University of Michigan, available from: soar-group@lists.sourceforge.net.

[29] Wikipedia search for “Soar (cognitive architecture)”, available at: http://en.wikipedia.org/wiki/Soar_(cognitive_architecture).

[30] Tambe, M., “Agent Architectures for Flexible, Practical Teamwork”, American Association of Artificial Intelligence, 1997.

[31] Lui, F., Vaughan, J., Jarvis, D., and Jarvis, J., “An Architecture to Support Autonomous Command Agents for OneSAF Testbed Simulations”, DSTO Land Operations Division and Agent Oriented Software Pty. Ltd., SimTect 2002.

[32] Agented Oriented Software website: http://www.agent-software.com.

[33] Howden, N., Ronnquist, R., Hodgson, A., and Lucas, A., “JACK Intelligent Agents – Summary of an Agent Infrastructure”, Agent Oriented Software Pty. Ltd., 2001, available at: http://users.cs.cf.ac.uk/O.F.Rana/agents2001/papers/18_howden.pdf.

[34] “JACK Intelligent Agents: Agent Manual”, Release 5.2, Agent Oriented Software Pty. Ltd., Carlton South, Victoria, Australia, June 2005.

[35] “JACK™ Intelligent Agents Teams Manual”, Release 5.2, Agent Oriented Software Pty. Ltd., Carlton South, Victoria, Australia, June 2005.

[36] Padgham, L., and Winikoff, M., “The Prometheus Methodology”, RMIT University, Melbourne, Australia, April 2004, available at: http://goanna.cs.rmit.edu.au/~linpa/Papers/bookchB.pdf.

[37] Karim, S., “Experiences with Prometheus and the Prometheus Design Tool (PDT)”, Deptartment of Information Systems, University of Melbourne, Australia, May 2004.

[38] Bordini, R., Hubner, J., and Tralamazza, D., “Using Jason to Implement a Team of Gold Miners”, available at: http://www.dur.ac.uk/r.bordini/Publications/UJITGMpd.pdf.

[39] Giunchiglia, F., Mylopoulos, J., and Perini, A., “The Tropos Software Development Methodology: Processes, Models and Diagrams”, Agent Oriented Software Engineering, Bologna, Italy, 2002.

[40] Mouratidis , H., Giorgini, P., Philp, I., and Manson, G., “Using Tropos Methodology to Model an Integrated Health Assessment System”, CEUR Workshop Proceedings, available at: http://ftp.informatik.rwth-aachen.de/Publications/CEUR-WS/Vol-57/id-16.pdf.

[41] Mouratidis, H., “Analyzing Security Requirements of Information Systems Using Tropos”, Innovative Informatics Group, School of Computing and Technology, University of East London, available at: http://www.uel.ac.uk/scot/ii/documents/SecureTroposPaper_finalVersion.pdf .

[42] Zambonelli, F., “Developing Multiagent Systems: The Gaia Methodology”, ACM Transactions on Software Engineering and Methodology, Vol. 12, No. 3, pp. 317–370, July 2003.

[43] Zhao, Q., Zhou, Y., and Perry, M., “Agent Design of SmArt License Management System Using Gaia Methodology”, Proceedings of the Third International Conference on Autonomic and Autonomous Systems, 2007.

[44] Casto, A., and Oliveira, E., “The Rationale Behind the Development of an Airline Operations Control Centre using GAIA-based Methodology”, to appear in Agent Oriented Software Engineering Journal, 2007.

[45] Shibghatullah, A., Eldabi, T., and Kuljis, J., “A Proposed MultiAgent Model for Bus Crew Scheduling”, Proceedings of the 2006 Winter Simulation Conference, 2006.

[46] Hill, R., Cheh, J., Gratch, J., Rosenbloom, P., and Tambe, M., “Intelligent Agents for the Synthetic Battlefield: A Company of Rotary Wing Aircraft”, Proceedings of Innovative Applications of Artificial Intelligence, 1997.

[47] Fletcher, M., “Cognitive Agent-based Approach to Varying Behaviours in Computer Generated Forces Systems to Model Scenarios like Coalitions”, Agent Oriented Software Limited, Distributed Intelligent Systems IEEE Workshop, June 2006.

[48] van Doesburg, W., Heuvelink, A., and van den Broek, E., “TACOP: A Cognitive Agent for a Naval Training Simulation Environment”, AAMAS 2005, Utrecht, Netherlands, 2005.

[49] Baxter, J., and Horn, G., “Controlling Teams of Uninhabited Air Vehicles”, QinetiQ Limited, AAMAS 2005, Utrecht, Netherlands, 2005.

[50] Lauren, M., and Stephen, R., “Map Aware Non-Uniform Automata Version 1.0 – User Manual”, produced for Land Operations Division, Defence Science and Technology Organisation, Australia, June 2001.

[51] Cioppa, T.M., and Lucas, T.W., “Military Application of Agent-Based Simulations”, Proceedings of the 2004 Winter Simulation Conference, 2004.

[52] Galligan, D., Galdorisi, G., and Marland, P., “Net Centric Maritime Warfare – Countering a ‘Swarm’ of Fast Inshore Attack Craft”. Paper presented at the 10th International Command and Control Research and Technology Symposium – The Future of C2, 2005, available at: http://www.dodccrp.org/events/10th_ICCRTS/CD/papers/053.pdf.

[53] Lauren, M., “How Agent Models Can Address Asymmetric Warfare: An ANZUS Collaboration”, Defence Technology Agency and New Zealand Defence Force, SimTect 2002.

[54] Randall, T.E., “Using Software Agents to Control the Behaviour of Simulated Entities”, DRDC Atlantic Technical Memorandum TM 2007-342, Defence R&D Canada – Atlantic, Dartmouth, Nova Scotia, Canada, December 2007.

List of symbols/abbreviations/acronyms/initialisms

ACL	Agent Communication Language
AI	Artificial Intelligence
BDI	Beliefs, Desires and Intentions
C2	Command and Control
DIS	Distributed Interactive Simulation
DRDC Atlantic	Defence R&D Canada - Atlantic
DTA	Defence Technology Agency
HIL	Human-in-the-Loop
HLA	High Level Architecture
JAL	JACK Agent Language
JDE	JACK Development Environment
JSAF	Joint Semi-Automated Forces
KE	Knowledge Engineering
KIF	Knowledge Interface Format
KQML	Knowledge Query and Manipulation Language
MANA	Map Aware Non-Uniform Automata
MAS	Multi-Agent System
OO	Object Oriented
OTB	OneSAF Testbed Baseline
PDT	Prometheus Design Tool
RE	Requirements Engineering
SA	Situational Awareness
SME	Subject Matter Expert
TACOP	Tactical Cognitive Opponent
TTCP	The Technical Cooperation Program
UAV	Unmanned Ariel Vehicle
VCS	Virtual Combat Systems
VMSA	Virtual Maritime Systems Architecture

This page intentionally left blank.

Distribution list

Document No.: DRDC Atlantic TM 2007-221

LIST PART 1: Internal Distribution by Centre

1	T.E. Randall
1	M.G. Hazen
1	H.J. Murphy
1	A.D. Gillis
1	G. Franck
1	H/MICS
3	DRDC Atlantic Library
9	TOTAL LIST PART 1

LIST PART 2: External Distribution by DRDKIM

1	DRDKIM
1	Library and Archives Canada, Attn: Military Archivist, Government Records Branch
2	TOTAL LIST PART 2

11 TOTAL COPIES REQUIRED

This page intentionally left blank.

DOCUMENT CONTROL DATA

(Security classification of title, body of abstract and indexing annotation must be entered when the overall document is classified)

1. ORIGINATOR (The name and address of the organization preparing the document. Organizations for whom the document was prepared, e.g. Centre sponsoring a contractor's report, or tasking agency, are entered in section 8.) Defence R&D Canada – Atlantic 9 Grove Street P.O. Box 1012 Dartmouth, Nova Scotia B2Y 3Z7	2. SECURITY CLASSIFICATION (Overall security classification of the document including special warning terms if applicable.) UNCLASSIFED (NON-CONTROLLED GOODS) DMC A REVIEW: GCEC JUNE 2010

3. TITLE (The complete document title as indicated on the title page. Its classification should be indicated by the appropriate abbreviation (S, C, R or U) in parentheses after the title.)

An Introduction to Software Agents

4. AUTHORS (last name, followed by initials – ranks, titles, etc. not to be used)

Randall, T.E.

5. DATE OF PUBLICATION (Month and year of publication of document.) February 2008	6a. NO. OF PAGES (Total containing information, including Annexes, Appendices, etc.) 66	6b. NO. OF REFS (Total cited in document.) 54

7. DESCRIPTIVE NOTES (The category of the document, e.g. technical report, technical note or memorandum. If appropriate, enter the type of report, e.g. interim, progress, summary, annual or final. Give the inclusive dates when a specific reporting period is covered.)

Technical Memorandum

8. SPONSORING ACTIVITY (The name of the department project office or laboratory sponsoring the research and development – include address.)

Defence R&D Canada – Atlantic
9 Grove Street
P.O. Box 1012
Dartmouth, Nova Scotia B2Y 3Z7

9a. PROJECT OR GRANT NO. (If appropriate, the applicable research and development project or grant number under which the document was written. Please specify whether project or grant.) 11bt	9b. CONTRACT NO. (If appropriate, the applicable number under which the document was written.)
10a. ORIGINATOR'S DOCUMENT NUMBER (The official document number by which the document is identified by the originating activity. This number must be unique to this document.) DRDC Atlantic TM 2007-221	10b. OTHER DOCUMENT NO(s). (Any other numbers which may be assigned this document either by the originator or by the sponsor.)

11. DOCUMENT AVAILABILITY (Any limitations on further dissemination of the document, other than those imposed by security classification.)

(X) Unlimited distribution
(   ) Defence departments and defence contractors; further distribution only as approved
(   ) Defence departments and Canadian defence contractors; further distribution only as approved
(   ) Government departments and agencies; further distribution only as approved
(   ) Defence departments; further distribution only as approved
(   ) Other (please specify):

12. DOCUMENT ANNOUNCEMENT (Any limitation to the bibliographic announcement of this document. This will normally correspond to the Document Availability (11). However, where further distribution (beyond the audience specified in (11) is possible, a wider announcement audience may be selected.))

Unlimited

13.	ABSTRACT (A brief and factual summary of the document. It may also appear elsewhere in the body of the document itself. It is highly desirable that the abstract of classified documents be unclassified. Each paragraph of the abstract shall begin with an indication of the security classification of the information in the paragraph (unless the document itself is unclassified) represented as (S), (C), (R), or (U). It is not necessary to include here abstracts in both official languages unless the text is bilingual.) This paper provides the reader with a general background on software agents, covering the main topics found in the agent literature. It consolidates pertinent information from a wide variety of sources into a single comprehensive document. Its aim is to provide enough information to educate those unfamiliar with agents, and guide them in deciding whether or not agents are an appropriate modelling tool for their own work.
14.	KEYWORDS, DESCRIPTORS or IDENTIFIERS (Technically meaningful terms or short phrases that characterize a document and could be helpful in cataloguing the document. They should be selected so that no security classification is required. Identifiers, such as equipment model designation, trade name, military project code name, geographic location may also be included. If possible keywords should be selected from a published thesaurus, e.g. Thesaurus of Engineering and Scientific Terms (TEST) and that thesaurus identified. If it is not possible to select indexing terms which are Unclassified, the classification of each should be indicated as with the title.) agents, behaviour modelling, synthetic forces